TLP **3** (2): 189–221, March 2003. © 2003 Cambridge University Press
DOI: 10.1017/S1471068402001527 Printed in the United Kingdom

Composing programs in a rewriting logic for declarative programming

J. M. MOLINA-BRAVO and E. PIMENTEL
Dpto. Lenguajes y Ciencias de la Computación, University of Málaga,
Campus de Teatinos, 29071 Málaga, Spain
(*e-mail:* {)*jmmb, ernesto*}*@lcc.uma.es*)

Abstract

Constructor-Based Conditional Rewriting Logic is a general framework for integrating first-order functional and logic programming which gives an algebraic semantics for non-deterministic functional-logic programs. In the context of this formalism, we introduce a simple notion of program module as an open program which can be extended together with several mechanisms to combine them. These mechanisms are based on a reduced set of operations. However, the high expressiveness of these operations enable us to model typical constructs for program modularization like hiding, export/import, genericity/instantiation, and inheritance in a simple way. We also deal with the semantic aspects of the proposal by introducing an immediate consequence operator, and studying several alternative semantics for a program module, based on this operator, in the line of logic programming: the operator itself, its least fixpoint (the least model of the module), the set of its pre-fixpoints (term models of the module), and some other variations in order to find a compositional and fully abstract semantics w.r.t. the set of operations and a natural notion of observability.

KEYWORDS: functional-logic programming, modules, compositionality, full abstraction, semantics

1 Introduction

Constructor-Based Conditional Rewriting Logic (CRWL)[1], presented in González-Moreno *et al.* (1999), is a quite general approach to declarative programming that combines (first-order) functional and logic paradigms by means of the notion of (possibly) *non deterministic lazy function*. The basic idea is that both relations and deterministic lazy functions are particular cases of non-deterministic lazy functions. This approach retains the advantages of deterministic functions while adding the possibility of modeling non-deterministic functions by means of *non-confluent constructor-based term rewriting systems*, where a given term may be rewritten to constructor terms (possibly with variables) in more than one way. Here a fundamental notion is that of *joinability*: two terms *a*,*b* are joinable iff they can be rewritten to a common – but not necessarily unique – constructor term. In González-Moreno

[1] CRWL must not be confused with the Rewriting Logic proposed in Meseguer (1992) as a unifying logical framework for concurrency. CRWL is a particular logic for dealing with indeterminism.

et al. (1999), CRWL is introduced with two equivalent proof calculi that govern deduction in this logic, an algebraic semantics for programs (theories) based on a freely generated model, and an operational semantics, based on a lazy narrowing calculus for solving goals, that is sound and complete w.r.t. the algebraic semantics.

Modularity is a central issue in all programming paradigms motivated by the need of mastering the complexity inherent in large programs. Modularity related with algebraic specifications (which, to some extent, can be viewed as a sort of first-order functional programming) has been extensively studied and all specification languages are extended for dealing with modules. In this field, a typical module consists of a body, an export interface, a list of imports and, possibly, a list of formal parameters, and typical operations with modules have to do with setting up hierarchical relationships between modules as the union of modules (with some constraints) and the application of a parameterized module to an actual module, and their semantics are given from a category-theoretic point of view (Goguen and Burstall, 1992; Ehrig and Mahr, 1990; Orejas, 1999). Nevertheless, there are other studies of modularity (Wirsing, 1990) with more flexible sets of operations semantically defined by means of operations on the sets of models, and also studies where modularity has been tackled with the tools of algebraic specifications, as in Bergstra *et al.* (1990), where an axiomatic specification is given for an algebra of non-parameterized modules and it is proved that each expression can be reduced to another one with, at most, an occurrence of the export (hiding) operator, and Durán (1999) where a constructive specification is given for an algebra of parameterized modules (without hiding) in Maude, and each expression is reduced to a flat module.

In the logic programming field, modularity has been the objective of different proposals – see Bugliesi *et al.* (1994) for a survey about the subject – which basically have followed two different guidelines. One, focused on programming-in-the-large, extends logic programming with modular constructs as a meta-linguistic mechanism (Brogi and Turini, 1995) and gives semantics to modules with the aid of the immediate consequence operator. And the other one, focused on programming-in-the-small, enriches the theory of Horn clauses with new logical connectives for dealing with modules (Miller, 1986). In the first line, there is the work Brogi (1993) where an algebra of logic programs is studied. This algebra is based on three basic operations (union, intersection and encapsulation) defined at the semantic level and then translate to the syntactic level. It is proved that each program expression is equivalent to a, possibly infinite, flat program, and also a transformation is defined for mapping program expressions into finite programs by introducing system generated predicates and adding a hidden part to each program. Notions of module hiding some predicates and module importation are built up with the aid of the basic operations.

On the other hand, in functional-logic programming we do not know any study of modularity semantically well founded. With this paper we have tried to contribute to filling this gap at least in the CRWL context. In this context, we deal with data constructors, as in logic programming, and functions defined by conditional rewrite rules, instead of predicates defined by Horn clauses, and we have proved (see section 3) that an operator, similar to the immediate consequence operator of logic programs, can be defined to each CRWL-program and its least fixpoint coincides with the freely generated term-model given in González-Moreno *et al.* (1999). All this has motivated our decision of developing a study of programs structuring and modularity in CRWL, based on a meta-linguistic mechanism, similar to the one which appears in Brogi (1993). However, we have defined an algebra of program modules based on a different set of operations (union, deletion of a signature of function symbols, closure w.r.t. a signature and renaming) defined at the syntactical level in such a way that each program expression can be reduced to a, possibly infinite, flat program. With these operations we can model as well as notions of module which hides some functions and module importation, module parameterization, instantiation and inheritance with overriding. Also, we have introduced a notion of protected signature labeling symbols with module expressions, which allows to define structured modules and a representation morphism that maps each program expression into a finite structured module. We use protected signature, not only for hiding functions as is done in Brogi (1993) for predicates, but also for hiding data constructors.

An important aspect to be considered when a language is extended for modular programming is the sound integration of the behavior of the modular operations into the semantics of the language. The compositionality of the semantics of a programming language is particularly relevant when modularity is involved. In fact, one of the most critical aspects in modular systems is the possibility of making a separate compilation of modules, and this can only be made in the presence of this property. On the other hand, full abstraction measures the implementation details of the semantics of a programming language. A non-fully abstract semantics makes the intended meaning of a program to include non relevant aspects, which do not depend on the behavior of the program but on a particular ‘implementation’. In some sense, full abstraction can be seen as the complementary property of compositionality, and the adequacy of a semantics is established when both full abstraction and compositionality are obtained. In Brogi (1993), the semantics of a program is given by its immediate consequence operator which captures the information concerning possible compositions, this semantics is compositional by construction and it is proved that also is fully abstract w.r.t. a notion of observable behavior given by the success sets of programs (least fixpoints of their immediate consequence operators). In CRWL-programming, the semantics given by the immediate consequence operator is compositional but not fully abstract when we take the freely generated term-model as observable behavior. For this reason, we study several alternative semantics to find one that is compositional and fully abstract.

We are confident that our work could serve as a reference to other studies of modularity in functional-logic programming, and, although we are focused on the modular aspects of the semantics, the results obtained in this paper, as well as the study of a wide range of other issues concerning semantics, makes the current work also relevant from a purely semantic point of view, in the context of rewriting logic-based programming languages. The approach to modularity in CRWL-programming, that we present here, substantially extends a previous one in Molina-Bravo and Pimentel (1997) with a more elaborate notion of program module and a new operation (renaming) that makes clear the difference between importation and instantiation, and a more recent one (Molina-Bravo, 2000) with the notions of structured module and module representation that allows to express closed modules by means of a finite number of rules and also to deal with local constructor symbols. For space reasons we do not include the proofs of the results; they are available at `http://arxiv.crg/abs/cs/0203006`.

The paper is organized as follows: in the next section we introduce the basic features of the CRWL approach to functional-logic programming and its model-theoretic semantics – for a detailed presentation we refer to González-Moreno *et al.* (1999). In section 3 we introduce an immediate consequence operator $\mathcal{T}_{\mathcal{R}}$, for each CRWL-program $\mathcal{R}$, and a fixpoint semantics that matches the free term-model $\mathcal{M}_{\mathcal{R}}$ proposed in González-Moreno *et al.* (1999). In section 4 we define a notion of (plain) module together with a reduced set of operations on program modules, and we express some modular constructions with these operations. In section 5 we give the $\mathcal{T}$-semantics that characterizes the meaning of a CRWL-program when we consider composition of programs and prove that this semantics is compositional but not fully abstract w.r.t. the set of operations, taking $\mathcal{M}_{\mathcal{R}}$ as the observable behavior of a program $\mathcal{R}$. In section 6 we introduce a fully abstract semantics by denoting a program module with the set of all its consistent term-models (pre-fixpoints of $\mathcal{T}_{\mathcal{R}}$); but this semantics is not compositional for the deletion of a signature. In section 7, we obtain a compositional and fully abstract semantics as an indexed family of sets of consistent term-models for single functions. In section 8, we introduce the notion of structured module as a finite representation of expressions made up from finite plain modules that allows the hiding of constructor symbols. Finally we present a discussion and some conclusions.

2 CRWL for declarative programming

A *signature with constructors* is a pair $\Sigma = (DS_\Sigma, FS_\Sigma)$, where DS_Σ and FS_Σ are countable disjoint sets of strings h/n with $n \in \mathbf{N}$. Each c such that $c/n \in DS_\Sigma$ is a *constructor symbol* with arity n and each f such that $f/n \in FS_\Sigma$ is a *(defined) function symbol* with arity n. The set of all constructor symbols and the set of all function symbols with arity n are denoted by DS^n_Σ and FS^n_Σ, respectively. Given a signature (with constructors) Σ and a set $\mathcal{V}$ of variable symbols, disjoint from all of the sets DS^n_Σ and FS^n_Σ, we define Σ-*terms* as follows: each symbol in $\mathcal{V}$ and each symbol in $DS^0_\Sigma \cup FS^0_\Sigma$ is a Σ-term, and for each $h \in DS^n_\Sigma \cup FS^n_\Sigma$ and $t_1, \ldots, t_n$ terms, $h(t_1, \ldots, t_n)$ is a term. $\mathbf{Term}_\Sigma$ is the set of all Σ-terms and $\mathbf{CTerm}_\Sigma$ the subset of those Σ-terms (called *constructor terms*) built up only with symbols in DS_Σ and $\mathcal{V}$. To cope with partial definition we add a new 0-arity constructor $\perp$ to each signature Σ obtaining an extended signature $\Sigma_\perp$ whose terms are called *partial* Σ-*terms*. When the signature Σ is clear, we will omit explicit mention of it, and we will write **Term** and **CTerm** (or $\mathbf{Term}_\perp$ and $\mathbf{CTerm}_\perp$ for $\Sigma_\perp$) respectively. Following the approach to non-determinism in Hussmann (1993), we only consider *C-substitutions* $\theta: \mathcal{V} \to \mathbf{CTerm}$. These mappings have natural extensions $\theta: \mathbf{Term} \to \mathbf{Term}$, also noted as θ, defined in the usual way, and the result of applying θ to the term t is written $t\theta$. Analogously, we define *partial C-substitutions* as mappings $\theta:\mathscr{V} \to \mathbf{CTerm}_\perp$. The set of all C-substitutions (partial C-substitutions) is written **CSubst** ($\mathbf{CSubst}_\perp$). A *signature morphism* $\rho:\Sigma \to \Sigma'$ from a signature $\Sigma = (DS_\Sigma, FS_\Sigma)$ to a signature $\Sigma' = (DS_{\Sigma'}, FS_{\Sigma'})$ consists of two mappings, that we denote with the same symbol $\rho:DS_\Sigma \to DS_{\Sigma'}$ and $\rho:FS_\Sigma \to FS_{\Sigma'}$, that map strings h/n into strings h'/n. By abuse of notation we will denote $h' = \rho(h)$. This allows us to define a mapping $\rho:\mathbf{Term}_{\Sigma_\perp} \to \mathbf{Term}_{\Sigma'_\perp}$ as $\rho(h) =_{def} h$, for $h \in \mathscr{V} \cup \{\perp\} \cup DS^0_\Sigma \cup FS^0_\Sigma$; $\rho(h(\bar{t})) =_{def} \rho(h)(\rho(t_1), \ldots, \rho(t_n))$, for $h \in DS^n_\Sigma \cup FS^n_\Sigma, n > 0$. $\rho(h) =_{def} h$, for $h \in \mathscr{V} \cup \{\perp\} \cup DS^0_\Sigma \cup FS^0_\Sigma$, and $\rho(h(\bar{t})) =_{def} \rho(h)\rho(t_1), \ldots, \rho(t_n))$, for $h \in DS^n_\Sigma \cup FS^n_\Sigma$ and $n > 0$. We will consider signature morphisms $\rho:\Sigma \to \Sigma$ such that $\rho(h/n) = h/n$ for every string h/n in DS_Σ. Such morphisms will be called *function symbol renamings*.

Given a signature Σ and a set $\mathscr{V}$ of variable symbols, there are two kinds of atomic *CRWL-formulas* for $a, b \in \mathbf{Term}_\perp$, *reduction statements* $a \to b$, with the intended meaning "a can be reduced to b," and *joinability statements* $a \bowtie b$, with the intended meaning "a and b can be reduced to a common value in **CTerm**". Terms $t \in \mathbf{CTerm}$ are intended to represent totally defined values whereas terms $t \in \mathbf{CTerm}_\perp$ represent partially defined values — to model the behavior of non-strict functions. Reduction statements $a \to t$ with $t \in \mathbf{CTerm}_\perp$, called *approximation* statements, have the intended meaning that t *approximates* a possible value of a, whereas $a \to t$ with $t \in \mathbf{CTerm}$ have the intended meaning that t *represents* a possible value of a – an expression may denote several values capturing the behavior of non-deterministic functions. Substitutions $\theta \in \mathbf{CSubst}_\perp$ and signature morphisms $\rho:\Sigma \to \Sigma'$ apply to formulas in the obvious way.

A *CRWL-program* is a CRWL-theory $\mathscr{R}$ defined as a signature Σ together with a set of *conditional rewrite rules* of the general form $f(\bar{t}) \to r \Leftarrow C$, where $f(\bar{t})$ is the left-hand side (lhs), r the right-hand side (rhs), C the condition of the rule, f is a function symbol with arity $n \geqslant 0$, and C consists of finitely many (possibly zero) joinability statements between fully defined terms (with no occurrence of $\perp$). When $n > 0$, $\bar{t}$ is a linear n-tuple (i.e. without repeated variables) of fully defined constructor terms $t_i \in \mathbf{CTerm}$. When $n = 0$ rules take the simpler form $f \to r \Leftarrow C$. Formal derivation of CRWL-statements from a given program $\mathscr{R}$ is governed by two equivalent calculi (see González-Moreno *et al.* (1999)). We present here the so-called Goal-Oriented Proof Calculus (GPC), which focuses on top-down proofs of reduction and joinability statements:

(Bo)	$e \to \perp,$	for $e \in \mathbf{Term}_\perp$;
(RR)	$e \to e,$	for $e \in \mathscr{V} \cup DS^0$;
(DS)	$\dfrac{e_1 \to t_1 \ldots e_n \to t_n}{c(\bar{e}) \to c(\bar{t})},$	for $c \in DS^n$ and $e_i, t_i \in \mathbf{Term}_\perp$;
(OR)	$\dfrac{e_1 \to t_1 \ldots e_n \to t_n \quad C \quad r \to t}{f(\bar{e}) \to t},$	if $(f(\bar{t}) \to r \Leftarrow C) \in [\mathscr{R}]_\perp$ and $t \not\equiv \perp$;
(Jo)	$\dfrac{a \to t \quad b \to t}{a \bowtie b},$	if $t \in \mathbf{CTerm}$ and $a, b \in \mathbf{Term}_\perp$;

where $[\mathscr{R}]_\perp = \{(l \to r \Leftarrow C)\theta \mid (l \to r \Leftarrow C) \in \mathscr{R},\ \theta \in \mathbf{CSubst}_\perp\}$ is the set of possibly partial *constructor instances* of rewrite rules and C-substitutions apply to rules in the obvious way. Rule **(Bo)** shows that a CRWL-reduction is related to the idea of approximation, and rule **(OR)** states that only constructor instances of rewrite rules are allowed in this calculus reflecting the so-called 'call-time-choice' (Hussmann, 1993) for non-determinism (values of arguments for functions are chosen before the call is made). When a reduction or joinability statement φ is derivable from a program $\mathcal{R}$ we write $\mathcal{R} \vdash_{CRWL} \varphi$ and we say that φ is *provable* in $\mathcal{R}$. *Goals* for a program $\mathcal{R}$ are finite conjunctions of atomic formulas, and *solutions* are C-substitutions that make goals derivable. In González-Moreno *et al.* (1999), a sound and complete lazy narrowing calculus for goal-solving can be found.

We interpret CRWL-programs over algebraic structures consisting of posets with bottom as carriers (i.e. sets D with a partial order $\sqsubseteq_D$ and a least element $\perp_D$), whose elements are thought of as finite approximations of possibly infinite values in the poset's ideal completion (Möller, 1985), and monotonic mappings from elements to cones (non-empty subsets of a poset with bottom, downclosed w.r.t. the partial order of the poset) as function symbol denotations reflecting possible non-determinism. Such a mapping $f : D \to \mathcal{C}(E)$ – where D, E are posets with bottom, and $\mathcal{C}(E)$ is the set of cones of E – can be extended to a monotonic mapping $\hat{f} : \mathcal{C}(D) \to \mathcal{C}(E)$, defined by $\hat{f}(C) =_{def} \bigcup_{u \in C} f(u)$ and also noted f by abuse of notation. In particular, deterministic function symbols are represented by mappings $f : D \to \mathcal{I}(E)$ computing directed cones or ideals (i.e., cones $\mathcal{C}$ such that for all $x, y \in \mathcal{C}$ there exists $z \in \mathcal{C}$ with $x \sqsubseteq z$ and $y \sqsubseteq z$) where $\mathcal{I}(E)$ is the set of ideals of E. These mappings become continuous mappings between algebraic cpos after performing the ideal completion (for a comprehensive exposition of these notions we refer to Abramsky and Jung (1994)). These ideas are behind the notion of CRWL-algebra.

Given a signature Σ and a set $\mathcal{V}$ of variable symbols, a *CRWL-algebra* of signature Σ is an algebraic structure $\mathcal{A} = (D_\mathcal{A}, \{c^\mathcal{A}\}_{c \in DS_\Sigma}, \{f^\mathcal{A}\}_{f \in FS_\Sigma})$ where the carrier $D_\mathcal{A}$ is a poset with bottom $\perp_\mathcal{A}$, $f^\mathcal{A}$ is a monotonic mapping $D^n_\mathcal{A} \to \mathcal{C}(D_\mathcal{A})$ for each $f \in FS^n_\Sigma$ and $c^\mathcal{A}$ is a monotonic mapping $D^n_\mathcal{A} \to \mathcal{I}(D_\mathcal{A})$ for each $c \in DS^n_\Sigma$. Both $f^\mathcal{A}$ and $c^\mathcal{A}$ reduce to cones when $n = 0$. To ensure preservation of finite and maximal elements in the ideal completion, we require for all $u_1, \ldots, u_n \in D_\mathcal{A}$ that there exists $v \in D_\mathcal{A}$ such that $c^\mathcal{A}(u_1, \ldots, u_n) = \langle v \rangle$, where $\langle v \rangle$ is the ideal generated by v (i.e. the set $\{d \in D_\mathcal{A} \mid d \sqsubseteq v\}$), and if all u_i are maximal (totally defined) then v must also be maximal. The class of all CRWL-algebras of signature Σ is denoted by $\mathbf{Alg}_\Sigma$. We are specially interested in *CRWL-term algebras*, which are CRWL-algebras with carrier $\mathbf{CTerm}_\perp$, ordered by the *approximation ordering* '$\sqsubseteq$', defined as the least partial ordering satisfying the following properties:

$$\begin{array}{lll} (a) & \perp \sqsubseteq t, & \forall t \in \mathbf{CTerm}_\perp; \\ (b) & c(\bar{s}) \sqsubseteq c(\bar{t}) & \text{if } s_i \sqsubseteq t_i,\ i = 1, \ldots, n, \text{ for } c \in DS^n_\Sigma, n \geqslant 0; \end{array}$$

and fixed interpretation for constructor symbols: $c^\mathcal{A} = \langle c \rangle$, for all $c \in DS^0_\Sigma$, and $c^\mathcal{A}(\bar{t}) = \langle c(\bar{t}) \rangle$, for all $c \in DS^n_\Sigma$ and $n \geqslant 0$. Therefore, two CRWL-term algebras of the same signature Σ will only differ in their interpretations for the function symbols of Σ. As a consequence of the above definition, for $s, t \in \mathbf{CTerm}_\perp$, $s \sqsubseteq t$ implies $s = \perp$ or $s = c(\bar{s})$ and $t = c(\bar{t})$ for some $c \in DS^n_\Sigma$ and $n \geqslant 0$ with each component $s_i \sqsubseteq t_i$. Also, for $s, t \in \mathbf{CTerm}_\perp$, $s \sqsubseteq t$ is equivalent to $\vdash_{CRWL} t \to s$. It can be proved, by induction, that every $\theta \in \mathbf{CSubst}_\perp$ is a monotonic mapping from $\mathbf{CTerm}_\perp$ to $\mathbf{CTerm}_\perp$, that is: $s \sqsubseteq t \;\Rightarrow\; s\theta \sqsubseteq t\theta$, for all $s, t \in \mathbf{CTerm}_\perp$.

A *valuation* over a structure $\mathscr{A} \in \mathbf{Alg}_\Sigma$ is any mapping $\eta : \mathscr{V} \to D_\mathscr{A}$. η is *totally defined* when $\eta(X)$ is maximal for all $X \in \mathscr{V}$. $\mathbf{Val}(\mathscr{A})$ is the set of all valuations over $\mathscr{A}$ and $\mathbf{DefVal}(\mathscr{A})$ the set of all totally defined valuations. Given a valuation η we can evaluate each partial Σ-term in $\mathscr{A}$ as follows:

$$\begin{array}{rlll}
[\![\, \perp \,]\!]^\mathscr{A}_\eta & =_{def} & \langle \perp_\mathscr{A} \rangle, & \\
[\![\, X \,]\!]^\mathscr{A}_\eta & =_{def} & \langle \eta(X) \rangle, & \forall X \in \mathscr{V}; \\
[\![\, c \,]\!]^\mathscr{A}_\eta & =_{def} & c^\mathscr{A}, & \forall c \in DS^0_\Sigma \cup FS^0_\Sigma; \\
[\![\, h(\overline{e}) \,]\!]^\mathscr{A}_\eta & =_{def} & \hat{h}^\mathscr{A}([\![\, e_1 \,]\!]^\mathscr{A}_\eta, \ldots, [\![\, e_n \,]\!]^\mathscr{A}_\eta), & \forall h \in DS^n_\Sigma \cup FS^n_\Sigma, n > 0.
\end{array}$$

In this way each partial Σ-term is evaluated to a cone. For each CRWL-algebra $\mathscr{A}$, every $\eta \in \mathbf{Val}(\mathscr{A})$, and $e \in \mathbf{Term}_\perp$, the following properties are proved in González-Moreno *et al.* (1999):

1. $[\![\, e \,]\!]^\mathscr{A}_\eta \in \mathscr{C}(D_\mathscr{A})$.
2. $[\![\, e \,]\!]^\mathscr{A}_\eta \in \mathscr{I}(D_\mathscr{A})$, if e is only built from deterministic functions (i.e. function symbols interpreted by ideal valued functions).
3. $[\![\, e \,]\!]^\mathscr{A}_\eta = \langle v \rangle$ for some $v \in D_\mathscr{A}$, if $e \in \mathbf{CTerm}_\perp$. Moreover, when $e \in \mathbf{CTerm}$ and $\eta \in \mathbf{DefVal}(\mathscr{A})$, v is maximal.
4. (Substitution Lemma) $[\![\, e\theta \,]\!]^\mathscr{A}_\eta = [\![\, e \,]\!]^\mathscr{A}_\rho$, for $\theta \in \mathbf{CSubst}_\perp$, where ρ is the uniquely determined valuation that satisfies $\langle \rho(X) \rangle = [\![\, X\theta \,]\!]^\mathscr{A}_\eta$, for all $X \in \mathscr{V}$.

From these results and taking into account that each substitution is equivalent to a valuation over any CRWL-term algebra, we have the following complementary results for term algebras:

Proposition 1
For each CRWL-term algebra $\mathscr{A}$ and every $\eta \in \mathbf{Val}(\mathscr{A})$ we have:

1. $[\![\, t \,]\!]^\mathscr{A}_\eta = \langle t\eta \rangle$ for every $t \in \mathbf{CTerm}_\perp$;
2. $[\![\, h(\overline{t}) \,]\!]^\mathscr{A}_\eta = h^\mathscr{A}(\overline{t}\eta)$ for all $h \in DS^n_\Sigma \cup FS^n_\Sigma$, $n > 0$, and $t_1, \ldots, t_n \in \mathbf{CTerm}_\perp$;
3. $[\![\, e\theta \,]\!]^\mathscr{A}_\eta = [\![\, e \,]\!]^\mathscr{A}_{\theta\eta}$ for all $e \in \mathbf{Term}_\perp$ and $\theta \in \mathbf{CSubst}_\perp$, where $\theta\eta$ represents the function composition $\eta \circ \theta$.

Models in CRWL are introduced from the following notion of satisfiability:

- $\mathscr{A}$ *satisfies a reduction statement* $a \to b$ under a valuation $\eta \in \mathbf{Val}(D_\mathscr{A})$, or $\mathscr{A} \models_\eta (a \to b)$, iff $[\![\, a \,]\!]^\mathscr{A}_\eta \supseteq [\![\, b \,]\!]^\mathscr{A}_\eta$.
- $\mathscr{A}$ *satisfies a joinability statement* $a \bowtie b$ under a valuation $\eta \in \mathbf{Val}(D_\mathscr{A})$, or $\mathscr{A} \models_\eta (a \bowtie b)$, iff $[\![\, a \,]\!]^\mathscr{A}_\eta \cap [\![\, b \,]\!]^\mathscr{A}_\eta$ contains a maximal element in $D_\mathscr{A}$.
- $\mathscr{A}$ *satisfies a rule* $l \to r \Leftarrow C$, or $\mathscr{A} \models (l \to r \Leftarrow C)$, iff $\mathscr{A} \models_\eta C$ implies $\mathscr{A} \models_\eta (l \to r)$, for every valuation $\eta \in \mathbf{Val}(D_\mathscr{A})$.
- $\mathscr{A}$ is a *model* of a program $\mathscr{R}$, i.e., $\mathscr{A} \models \mathscr{R}$, iff $\mathscr{A}$ satisfies all rules in $\mathscr{R}$.

CRWL-provability is sound and complete w.r.t. this model-theoretic semantics when we consider totally defined valuations only. In González-Moreno *et al.* (1999) it is proved that for any program $\mathscr{R}$ and any approximation or joinability statement φ, $\mathscr{R} \vdash_{CRWL} \varphi$ is equivatent to $\mathscr{A} \models_\eta \varphi$, for every $\mathscr{A}$ model of $\mathscr{R}$ and $\eta \in \mathbf{DefVal}(D_\mathscr{A})$.

This result is achieved with the help of a CRWL-term algebra $\mathscr{M}_{\mathscr{R}}$ characterized by the following interpretation for any defined function symbol $f \in FS^n_\Sigma$, $n \geqslant 0$,

$$f^{\mathscr{M}_{\mathscr{R}}}(\bar{t}) =_{def} \{r \in \mathbf{CTerm}_\perp \mid \mathscr{R} \vdash_{CRWL} f(\bar{t}) \to r\}.$$

$\mathscr{M}_{\mathscr{R}}$ is such that $\mathscr{R} \vdash_{CRWL} \varphi \;\Leftrightarrow\; \mathscr{M}_{\mathscr{R}} \models_{id} \varphi$ for any approximation or joinability statement φ. According to this result, $\mathscr{M}_{\mathscr{R}}$ is taken as the *canonical model* of the program $\mathscr{R}$. Also, in González-Moreno *et al.* (1999), it is proved that this model is freely generated by $\mathscr{V}$ in the category of all models of $\mathscr{R}$. This is the *model-theoretical semantics* of the program $\mathscr{R}$.

Given a signature Σ and a function symbol renaming $\rho : \Sigma \to \Sigma$, for each CRWL-term algebra $\mathscr{A} = (\mathbf{CTerm}_\perp, \{c^{\mathscr{A}}\}_{c \in DS_\Sigma}, \{f^{\mathscr{A}}\}_{f \in FS_\Sigma})$ of this signature we can define another CRWL-term algebra $\mathscr{A}_\rho = (\mathbf{CTerm}_\perp, \{c^{\mathscr{A}_\rho}\}_{c \in DS_\Sigma}, \{f^{\mathscr{A}_\rho}\}_{f \in FS_\Sigma})$, such that $f^{\mathscr{A}_\rho} = \rho(f)^{\mathscr{A}}$. The relation between evaluation and satisfaction in $\mathscr{A}$ and evaluation and satisfaction in $\mathscr{A}_\rho$ is stated by the following proposition.

Proposition 2

Given a signature Σ, for every CRWL-term algebra $\mathscr{A}$ of this signature, every function symbol renaming $\rho : \Sigma \to \Sigma$, and all $\theta \in \mathbf{CSubst}_\perp$, we have

1. $(\rho(t))\theta = \rho(t\theta)$, for all $t \in \mathbf{Term}_\perp$.
2. $[\![\, \rho(t) \,]\!]^{\mathscr{A}}_\theta = [\![\, t \,]\!]^{\mathscr{A}_\rho}_\theta$, for all $t \in \mathbf{Term}_\perp$.
3. $\mathscr{A} \models_\theta \rho(\varphi) \;\Leftrightarrow\; \mathscr{A}_\rho \models_\theta \varphi$, for any reduction or joinability statement φ.

3 Fixpoint semantics

In this section we prove, for every CRWL-program $\mathscr{R}$, that $\mathscr{M}_{\mathscr{R}}$ is the least fixpoint of an operator defined over CRWL-term algebras. The approach we use here is similar to that applied in the field of logic programming (Apt, 1990). However, the notion of interpretation, and the corresponding mathematical aspects, have to be reformulated in the context of CRWL-term algebras. This approach has been also used in González-Moreno (1994) in the context of a previous formalism to model functional-logic programming. However, this work does not deal with some relevant aspects (e.g. non-determinism) of the CRWL-programming version we are considering here.

Let $\mathbf{TAlg}_\Sigma$ be the set of all CRWL-term algebras of a signature Σ associated to a CRWL-program $\mathscr{R}$. We can define the relationship $\mathscr{A} \sqsubseteq \mathscr{B}$, between two algebras $\mathscr{A}, \mathscr{B} \in \mathbf{TAlg}_\Sigma$, as $f^{\mathscr{A}}(\bar{t}) \subseteq f^{\mathscr{B}}(\bar{t})$ for all $f \in FS^n_\Sigma$, when $n > 0$, and $f^{\mathscr{A}} \subseteq f^{\mathscr{B}}$, when $n = 0$. This relationship is obviously a partial ordering and $(\mathbf{TAlg}_\Sigma, \sqsubseteq)$ is a poset. This poset has a bottom $\perp_\Sigma$ and a top $\top_\Sigma$ characterized by $f^{\perp_\Sigma}(\bar{t}) = \langle \perp \rangle$ and $f^{\top_\Sigma}(\bar{t}) = \mathbf{CTerm}_\perp$, respectively, for each $f \in FS^n_\Sigma$ and $n \geqslant 0$. Given a subset $\mathbf{S} \subseteq \mathbf{TAlg}_\Sigma$, the following definitions: $f^{\sqcup \mathbf{S}}(\bar{t}) =_{def} \bigcup_{\mathscr{A} \in \mathbf{S}} f^{\mathscr{A}}(\bar{t})$, and $f^{\sqcap \mathbf{S}}(\bar{t}) =_{def} \bigcap_{\mathscr{A} \in \mathbf{S}} f^{\mathscr{A}}(\bar{t})$, for each $f \in FS^n_\Sigma$ and $n \geqslant 0$, characterize two CRWL-term algebras, $\sqcup \mathbf{S}$ and $\sqcap \mathbf{S}$ respectively, because the union and intersection of any number of cones are cones also, and the resulting functions in the above definitions are obviously monotonic if $f^{\mathscr{A}}$ is monotonic for all $\mathscr{A} \in \mathbf{S}$. Clearly, $\sqcup \mathbf{S}$ and $\sqcap \mathbf{S}$ are the least upper bound and the greatest lower bound of $\mathbf{S}$, respectively. So, $(\mathbf{TAlg}_\Sigma, \sqsubseteq)$ is a complete lattice.

Valuations (substitutions) of terms in term algebras can be considered continuous mappings from algebras to cones in the sense given by the following lemma.

Lemma 1 (*Continuity of valuations in* $\mathbf{TAlg}_\Sigma$)
For each term $e \in \mathbf{Term}_\perp$ and each substitution $\theta \in \mathbf{CSubst}_\perp$

1. $\mathscr{A} \sqsubseteq \mathscr{B} \Rightarrow [\![e]\!]^{\mathscr{A}}_\theta \subseteq [\![e]\!]^{\mathscr{B}}_\theta$, for $\mathscr{A}, \mathscr{B} \in \mathbf{TAlg}_\Sigma$.
2. $[\![e]\!]^{\sqcup \mathbf{D}}_\theta = \bigcup_{\mathscr{A} \in \mathbf{D}} [\![e]\!]^{\mathscr{A}}_\theta$, for all directed subsets $\mathbf{D} \subseteq \mathbf{TAlg}_\Sigma$.

Another interesting result relates satisfiability of joinability statements in the least upper bound of a directed set of term algebras with satisfiability in, at least, one of the algebras of the set.

Lemma 2
Let C be a finite set of joinability statements and $\mathbf{D}$ a directed subset of $\mathbf{TAlg}_\Sigma$, then $\sqcup \mathbf{D} \models_\theta C$ implies that there exists $\mathscr{A} \in \mathbf{D}$ such that $\mathscr{A} \models_\theta C$.

Given a CRWL-program $\mathscr{R}$, with a signature Σ, we can define an algebra transformer $\mathscr{T}_\mathscr{R} : \mathbf{TAlg}_\Sigma \to \mathbf{TAlg}_\Sigma$, similar to the immediate consequences operator used in logic programming, by fixing the interpretation of each function symbol $f \in FS^n_\Sigma$, in a transformed algebra $\mathscr{T}_\mathscr{R}(\mathscr{A})$, as the result of one step applications of reduction statements corresponding to instances – not necessarily ground – of those rules of $\mathscr{R}$, defining f, satisfied in $\mathscr{A}$. We formalize this idea defining, for each $f \in FS^n_\Sigma$, $n \geqslant 0$,

$$f^{\mathscr{T}_\mathscr{R}(\mathscr{A})}(\bar{t}) =_{def} \{t \mid \exists (f(\bar{s}) \to r \Leftarrow C) \in [\mathscr{R}]_\perp, s_i \sqsubseteq t_i, \mathscr{A} \models_{id} C, t \in [\![r]\!]^{\mathscr{A}}_{id}\} \cup \{\perp\},$$

that is basically a union of cones $[\![r]\!]^{\mathscr{A}}_{id}$. This definition corresponds to a monotonic mapping because all rule instances $(f(\bar{s}) \to r \Leftarrow C) \in [\mathscr{R}]_\perp$, applicable to arguments $\bar{t'}$ are also applicable to arguments $\bar{t}$ such that $t'_i \sqsubseteq t_i$, for $i = 1, \ldots, n$, and so the corresponding interpretation characterizes a CRWL-term algebra. From this definition of $\mathscr{T}_\mathscr{R}$ we can derive the continuity of the operator in $\mathbf{TAlg}_\Sigma$.

Proposition 3
For each program $\mathscr{R}$ its associated operator $\mathscr{T}_\mathscr{R}$ is continuous.

Thus, $\mathscr{T}_\mathscr{R}$ has a least fixpoint $\mathscr{F}_\mathscr{R}$ given by $\sqcup \mathbf{A}_\mathscr{R}$ (that is also the least pre-fixpoint), where $\mathbf{A}_\mathscr{R}$ is the chain of CRWL-term algebras $\mathscr{A}_i, i \in \mathbf{N}$, such that $\mathscr{A}_0 = \perp_\Sigma \sqsubseteq \cdots \sqsubseteq \mathscr{A}_{i+1} = \mathscr{T}_\mathscr{R}(\mathscr{A}_i) \sqsubseteq \cdots$. $\mathscr{F}_\mathscr{R}$ is also denoted as ${\mathscr{T}_\mathscr{R}}^\omega(\perp_\Sigma)$ (see Abramsky and Jung (1994)). To prove that $\mathscr{F}_\mathscr{R}$ coincides with $\mathscr{M}_\mathscr{R}$ we need two lemmata, one characterizing the set of term models and other relating CRWL-provability with $\mathbf{A}_\mathscr{R}$ satisfiability.

Lemma 3 (*Model characterization*)
Given a program $\mathscr{R}$, $\mathscr{M}$ is a term model for $\mathscr{R}$ iff $\mathscr{T}_\mathscr{R}(\mathscr{M}) \sqsubseteq \mathscr{M}$

Lemma 4
Given $e \in \mathbf{Term}_\perp$ and $t \in \mathbf{CTerm}_\perp$, we have that $\mathscr{R} \vdash_{CRWL} e \to t$ implies $\mathscr{A}_i \models_{id} e \to t$ for some $\mathscr{A}_i \in \mathbf{A}_\mathscr{R}$.

From the above results we obtain the following proposition.

Proposition 4
For every program $\mathscr{R}$, $\mathscr{M}_{\mathscr{R}}$ is the least fixpoint (and the least pre-fixpoint) of $\mathscr{T}_{\mathscr{R}}$.

Thus, if we consider the meaning of a program $\mathscr{R}$ as the least fixpoint of its associated transformer $\mathscr{T}_{\mathscr{R}}$, then this fixpoint semantics coincides with the model-theoretic semantics as it happens in logic programming. In fact, this semantics would correspond to the C-semantics in Falaschi *et al.* (1993).

Definition 1 (*Least model semantics*)
For each program $\mathscr{R}$ we define its least model semantics as: $\{[\mathscr{R}]\}_{LM} =_{def} \mathscr{M}_{\mathscr{R}}$.

4 An algebra of CRWL-program modules

For designing large programs it is convenient to separate the whole task into subtasks of manageable size and construct programs in a structured fashion by combining and modifying smaller programs. This idea has been extended to many programming languages giving rise to different notions of program module, each one being attached to a programming paradigm. In CRWL-programming we are going to follow an approach close to that developed in Brogi (1993) for logic programming, where modules are open programs in the sense that function definitions in a module can be completed with definitions for the same functions in other modules. We will consider a global signature with bottom $\Sigma_\perp = (DS_{\Sigma_\perp}, FS_{\Sigma_\perp})$ and a countable set $\mathscr{V}$ of variable symbols and will construct modules and module expressions with symbols of these sets. $\Sigma_\perp$ and $\mathscr{V}$ will characterize the environment where modules are written. Also we will consider all constructor symbols in $DS_{\Sigma_\perp}$ common to all program modules as it is usual in other proposals of modularity for declarative programming, like Brogi *et al.* (1994) and Orejas *et al.* (1997), where compositionality and full abstraction are dealt with. With this decision we give up any possibility of data abstraction and the only contribution of a program module to the environment will be a set of (definition) rules for a subsignature of function symbols. We will take this subsignature to denote the exportable resources of the module, and the set of rules as its body. In a program module, function symbols may appear – in the rhs of a rule – with no definition rule in this module. Although it may be assumed that all function symbols are defined in each program module by assuming an implicit rule $f(\bar{t}) \to \perp$ for each function symbol f with no definition rule, these symbols will be assumed to be provided by other modules and they will be taken to denote the resources that have to be imported. They will be the parameters of the module. From these considerations we propose the following definition for the notion of module in CRWL-programming

Definition 2 (*Module*)
A *module* in CRWL-programming is a tuple $< \sigma_p, \sigma_e, \mathscr{R} >$ where $\mathscr{R}$ is a set of program rules $f(\bar{t}) \to r \Leftarrow C$ $(r \neq \perp)$, σ_e is the (exported) signature of function symbols with a definition rule in $\mathscr{R}$, σ_p is the (parameter) signature of those function symbols with no definition rule in $\mathscr{R}$ that appear in any rule (i.e. they are invoked but not defined).

$\mathscr{R}$ is the *body* of the module and (σ_p, σ_e) its *interface*. The interface of a module could be inferred from its body if one knows which are the constructor symbols. However, as we consider all constructor symbols common to all program modules, we do not include an explicit declaration of these symbols in any module and have to make explicit parameter signatures in order to distinguish between function and constructor symbols. In this way, every symbol not occurring in σ_e nor σ_p will be a constructor symbol. Next, we have an example of a module definition.

Example 3
This example shows a module for constructing ordered lists of natural numbers with functions for inserting elements, checking the type of an element, and compare natural numbers.

```
OrdNatList =
  < {},                               % Parameter signature
    {isnat/1, leq/2, insert/2},       % Exported signature
    { isnat(zero)    -> true.
      isnat(succ(X)) -> isnat(X).
      leq(zero,zero)       -> true.
      leq(zero,succ(X))    -> isnat(X).
      leq(succ(X),zero)    -> false <= isnat(X) >< true.
      leq(succ(X),succ(Y)) -> leq(X,Y).
      insert(X,[])     -> [X]              <= isnat(X) >< true.
      insert(X,[Y|Ys]) -> [X|[Y|Ys]]       <= leq(X,Y) >< true.
      insert(X,[Y|Ys]) -> [Y|insert(X,Ys)] <= leq(X,Y) >< false.}>
```

In this module the parameter signature is empty, and symbols like `zero/0`, `succ/1`, `[]/0`, `[_|_]/2` with no definition rule are considered constructor symbols, because they are not included in the parameter signature (and obviously because they occur in arguments of left-hand sides).

We write $\mathbf{PMod}(\Sigma_\perp)$ for the class of all program modules which can be defined with a signature $\Sigma_\perp$, $\mathbf{SubSig}(\Sigma_\perp)$ for the set of all subsignatures of a signature $\Sigma_\perp$, and $\mathbf{Prg}(\Sigma_\perp)$ for the class of all sets of rules (programs) which can be defined with $\Sigma_\perp$. On $\mathbf{PMod}(\Sigma_\perp)$ we define three projections:

- $par : \mathbf{PMod}(\Sigma_\perp) \rightarrow \mathbf{SubSig}(\Sigma_\perp)$ such that $par(<\sigma_p, \sigma_e, \mathscr{R}>) = \sigma_p$,
- $exp : \mathbf{PMod}(\Sigma_\perp) \rightarrow \mathbf{SubSig}(\Sigma_\perp)$ such that $exp(<\sigma_p, \sigma_e, \mathscr{R}>) = \sigma_e$, and
- $rl : \mathbf{PMod}(\Sigma_\perp) \rightarrow \mathbf{Prg}(\Sigma_\perp)$ such that $rl(<\sigma_p, \sigma_e, \mathscr{R}>) = \mathscr{R}$,

which give respectively the parameter signature, the exported signature, and the body of a module.

4.1 Basic operations on modules

In this section we present a set of basic operations with modules that allows us to express typical features of modularization techniques such as information hiding/abstraction, import/export relationships and inheritance related to function symbols as is done in Brogi (1993), but our set of operations is different and we give syntactic definitions for it. We use three operations: union of programs, closure w.r.t. a signature and deletion of a signature, that are sufficient to express the most extended ways of composing modules and their relationships, and we do not need the intersection of programs, used in Brogi (1993) to model hiding, because we directly deal with signatures in the closure. To give more flexibility in expressing importation and instantiation, we also include a renaming operation. We define our operations in such a way that all module expressions can be reduced to a flat module $< \sigma_p, \sigma_e, \mathscr{R} >$ – where $\mathscr{R}$ could be an infinite set of rules. This is something like a *presentation semantics* (Wirsing, 1990).

First we define the *union* of two modules as the module obtained as the simple union of signatures and rules.

Definition 4 (*Union*)
Given two modules $\mathscr{P}_1 =< \sigma_p^1, \sigma_e^1, \mathscr{R}_1 >$ and $\mathscr{P}_2 =< \sigma_p^2, \sigma_e^2, \mathscr{R}_2 >$, their union $\mathscr{P}_1 \cup \mathscr{P}_2$ is defined as the module $< (\sigma_p^1 \cup \sigma_p^2) \setminus (\sigma_e^1 \cup \sigma_e^2), \sigma_e^1 \cup \sigma_e^2, \mathscr{R}_1 \cup \mathscr{R}_2 >$.

Each argument in this operation is considered an open program that can be extended or completed with the other argument possibly with additional rules for its exported function symbols.

Example 5
Let us consider the following module with a function to give change for an amount of money. Values for coins are provided by the non-deterministic function `coin/0`, whereas `getcoin/1` gives different possibilities to get a coin for a fixed amount. Finally, the function `change/1` returns a list with the coins corresponding to the change. In this example, we are assuming a predefined arithmetic with the usual notation for natural numbers. This was not the case in Example 3.

```
MoneyChange =
  < {_=<_/2, _-_/2},
    {coin/0,getcoin/1,change/1},
    { coin -> 1. coin -> 5. coin -> 10.
      getcoin(N) -> C <= coin >< C, C =< N >< true.
      change(0) -> [].
      change(N) -> [C|change(N-C)] <= getcoin(N) >< C. } >
```

We can extend this module with another module for providing new coins:

```
NewCoins = <{},{coin/0},{coin -> 15. coin -> 20.}>
```

simply by joining them to obtain

```
MoneyChange ∪ NewCoins =
  < {_=<_/2, _-_/2},
    {coin/0,getcoin/1,change/1},
    { coin -> 1. coin -> 5. coin -> 10. coin -> 15. coin -> 20.
      getcoin(N) -> C <= coin >< C, C =< N >< true.
      change(0) -> [].
      change(N) -> [C|change(N-C)] <= getcoin(N) >< C. } >
```

Union of modules is idempotent, associative, commutative, and there exists a null element: the module $\mathscr{O} =< \sigma_o, \sigma_o, \emptyset >$, where σ_o is the empty signature of function symbols, representing the module with no rule.

Proposition 5
The union of modules has the following properties:

1. $\mathscr{P} \cup \mathscr{O} = \mathscr{P}$, for every module $\mathscr{P}$.
2. $\mathscr{P} \cup \mathscr{P} = \mathscr{P}$, for every module $\mathscr{P}$.
3. $(\mathscr{P} \cup \mathscr{P}_1) \cup \mathscr{P}_2 = \mathscr{P} \cup (\mathscr{P}_1 \cup \mathscr{P}_2)$, for all modules $\mathscr{P}$, $\mathscr{P}_1$ and $\mathscr{P}_2$.
4. $\mathscr{P}_1 \cup \mathscr{P}_2 = \mathscr{P}_2 \cup \mathscr{P}_1$, for all modules $\mathscr{P}_1$ and $\mathscr{P}_2$.

The second operation is the *closure* of a module *w.r.t. a given signature* σ. This operation makes accessible the signature σ in an extensional way (i.e. only provable approximations can be used) and hides the rest. To define this operation, we need to introduce the notion of canonical rewrite rule.

Definition 6 (*Canonical rewrite rule*)
Given a term $f(\bar{t})$, with $f \in FS^n_\Sigma$ and each $t_i \in \mathbf{CTerm}_\perp$, and $r \in \mathbf{CTerm}_\perp$, we define the canonical rewrite rule $crr(f(\bar{t}), r)$ which reduces $f(\bar{t})$ to r, as the rule $f(\bar{t}') \rightarrow r \Leftarrow C$, constructed by substituting in $\bar{t}$ each occurrence of a repeated variable X or $\perp$ with a fresh variable Y and adding in C a joinability statement $X \bowtie Y$ for each occurrence of a repeated variable X, and a statement $X \bowtie X$ for each variable X in r and each variable with only one occurrence in $\bar{t}$.

In this way we obtain a program rule (with $\bar{t}'$ linear and each $t'_i \in \mathbf{CTerm}$) from which $f(\bar{t}) \rightarrow r$ can be proved, because for $\theta_{\bar{t}} \in \mathbf{CSubst}_\perp$ such that $\theta_{\bar{t}}(Y) = X$ for each fresh variable Y that substitutes an occurrence of X in $\bar{t}$, $\theta_{\bar{t}}(Y) = \perp$ for each fresh variable Y that substitutes an occurrence of $\perp$, and $\theta_{\bar{t}}(X) = X$ for all other variables, $C\theta_{\bar{t}}$ always can be proved and $(f(\bar{t}') \rightarrow r)\theta_{\bar{t}}$ is $f(\bar{t}) \rightarrow r$.

Example 7
The canonical rewrite rule which reduces $f(\perp, b(X,Y), X)$ to $a(X,Z)$ is:

$$f(V, b(X,Y), X1) \rightarrow a(X,Z) \quad \Leftarrow \quad \{X1 \bowtie X, Y \bowtie Y, Z \bowtie Z\},$$

and the associated substitution $\theta_{\bar{t}}$ is such that $\theta_{\bar{t}}(X1) = X$, $\theta_{\bar{t}}(V) = \perp$, and $\theta_{\bar{t}}(W) = W$ for all other variables W. In this case $C\theta_{\bar{t}} = \{X \bowtie X, Y \bowtie Y, Z \bowtie Z\}$ and all these joinability statements can be trivially derived from (**RR**) and (**Jo**), and therefore $f(\perp, b(X,Y), X) \rightarrow a(X,Z)$ by the (**OR**) rule.

Now, we can define the closure of a module as follows.

Definition 8 (*Closure w.r.t. a signature*)
Given a module $\mathscr{P} =< \sigma_p, \sigma_e, \mathscr{R} >$, its closure $\overline{\mathscr{P}}^\sigma$ w.r.t. a signature of function symbols σ is defined as the module:

$$< \sigma_o, \sigma'_e, \{crr(f(\bar{t}), r) \mid f/n \in \sigma,\ r \in \mathbf{CTerm}_\perp,\ r \neq \perp,\ \mathscr{R} \vdash_{CRWL} f(\bar{t}) \rightarrow r\} >,$$

where σ_o denotes the empty signature of function symbols, $t_i \in \mathbf{CTerm}_\perp$ for each component of the tuple $\bar{t}$, and σ'_e is the corresponding exported signature.

The closure of a module is a module with a possibly infinite set of rules (although the exported signature is always finite) equivalent to the union of the graphs in $\mathscr{M}_\mathscr{P}$ of all functions defined in $\mathscr{P}$ and contained in σ. Note that $\sigma'_e \subseteq \sigma_e \cap \sigma$ because a function in $\sigma_e \cap \sigma$ that depends on functions in the parameter signature could remain with no definition rule – or with the only rule $f(\bar{t}) \to \bot$ – after closing the module. As a syntactic simplification we will write $\overline{\mathscr{P}}$ instead of $\overline{\mathscr{P}}^{\sigma_e}$ for each module $\mathscr{P} =< \sigma_p, \sigma_e, \mathscr{R} >$.

Example 9
Let us consider the following module about week days, where two functions are defined to get the `next` day and the day `before` of a given day.

```
WeekDays = < {},
             {next/1,before/1},
             { next(mo) -> tu.   next(tu) -> we.   next(we) -> th.
               next(th) -> fr.   next(fr) -> sa.   next(sa) -> su.
               next(su) -> mo.
               before(X) -> Y <= next(Y) >< X. }                    >
```

The closure of this module w.r.t. its whole exported signature is the module

$\overline{\texttt{WeekDays}}$
```
           = < {},
             {next/1,before/1},
             { next(mo) -> tu.   next(tu) -> we.   next(we) -> th.
               next(th) -> fr.   next(fr) -> sa.   next(sa) -> su.
               next(su) -> mo.
               before(tu) -> mo.  before(we) -> tu.  before(th) -> we.
               before(fr) -> th.  before(sa) -> fr.  before(su) -> sa.
               before(mo) -> su. }         >
```

Closure w.r.t. a signature is in some way the counterpart of the encapsulation operation '*' in Brogi (1993), but it is more general because it has a twofold effect: hiding all rules in the module and restricting the visible signature, so we need no intersection of modules – as is needed in Brogi (1993) – to restrict visibility in a closed module. Variables and bottom can appear in the rules of a closed module, but no functions in the parameter signature.

Proposition 6
Closure of modules has the following properties, where σ, σ_1 and σ_2 are signatures of function symbols,

1. $\overline{\mathscr{P}}^{\sigma} = \mathscr{O}$, for every module $\mathscr{P}$ and every signature σ such that $\sigma \cap exp(\mathscr{P}) = \sigma_o$.
2. $\overline{\mathscr{O}}^{\sigma} = \mathscr{O}$, for every signature σ and the null module $\mathscr{O}$.
3. $\overline{\mathscr{P}}^{\sigma_1 \cup \sigma_2} = \overline{\mathscr{P}}^{\sigma_1} \cup \overline{\mathscr{P}}^{\sigma_2}$, for every module $\mathscr{P}$ and signatures σ_1, σ_2.
4. $\overline{\overline{\mathscr{P}}^{\sigma_1}}^{\sigma_2} = \overline{\mathscr{P}}^{\sigma_1 \cap \sigma_2} = \overline{\overline{\mathscr{P}}^{\sigma_2}}^{\sigma_1}$, for every module $\mathscr{P}$ and signatures σ_1, σ_2.
5. $\overline{\mathscr{P}_1 \cup \mathscr{P}_2}^{\sigma} = \overline{\mathscr{P}_1}^{\sigma} \cup \overline{\mathscr{P}_2}^{\sigma}$, for modules $\mathscr{P}_1$ and $\mathscr{P}_2$ defining disjoint signatures and such that neither $\mathscr{P}_1$ nor $\mathscr{P}_2$ use the signature defined in the other module.

Our third operation is the *deletion of a signature* in a module.

Definition 10 (*Deletion of a signature*)
Given a module $\mathscr{P} =< \sigma_p, \sigma_e, \mathscr{R} >$, the deletion in $\mathscr{P}$ of a signature of function symbols σ produces the module $\mathscr{P} \setminus \sigma =_{def} < \sigma'_p, \sigma_e \setminus \sigma, \mathscr{R} \setminus \sigma >$, where $\mathscr{R} \setminus \sigma$ denotes the set of those rules in $\mathscr{R}$ defining function symbols not appearing in σ, and σ'_p denotes the corresponding parameter signature.

We do not give an explicit expression for $par(\mathscr{P} \setminus \sigma)$ in terms of $par(\mathscr{P})$ because new parameters can appear and old ones can disappear with the deletion of rules in $rl(\mathscr{P})$. However, $par(\mathscr{P} \setminus \sigma) \subseteq \sigma_p \cup (\sigma_e \cap \sigma)$ is satisfied.

Example 11
In the module `OrdNatList` of Example 3 we can delete or abstract the signature `{isnat/1,leq/2}` to obtain the following parameterized module

```
OrdNatList\{isnat/1,leq/2} =
  < {isnat/1,leq/2},
    {insert/2},
    { insert(X,[])     -> [X]             <= isnat(X) >< true.
      insert(X,[Y|Ys]) -> [X|[Y|Ys]]      <= leq(X,Y) >< true.
      insert(X,[Y|Ys]) -> [Y|insert(X,Ys)] <= leq(X,Y) >< false. } >
```

The resulting module is now parameterized by the two symbol functions `isnat/1` and `leq/2`, whereas only the function `insert/2` is exported.

This operation recalls the *undefine* clause in the object-oriented language Eiffel, and we will use it (combined with the union) to perform inheritance with overriding. Note the differences between the deletion of a signature and the closure w.r.t. a signature. The former operation removes rules defining function symbols in the signature – but not those rules containing invocations in their rhs or condition – whereas the latter only hides the definitions of the functions in the signature, but maintains their consequences – hiding all other functions.

Proposition 7
The deletion of a signature (of function symbols) in a module has the following properties, where σ, σ_1 and σ_2 are signatures of function symbols,

1. $\mathscr{P} \setminus \sigma = \mathscr{O}$, for every module $\mathscr{P}$ and every σ such that $exp(\mathscr{P}) \subseteq \sigma$.
2. $\mathscr{P} \setminus \sigma = \mathscr{P}$, for every module $\mathscr{P}$ and every σ such that $exp(\mathscr{P}) \cap \sigma = \sigma_o$.
3. $(\mathscr{P} \setminus \sigma_1) \setminus \sigma_2 = \mathscr{P} \setminus (\sigma_1 \cup \sigma_2) = (\mathscr{P} \setminus \sigma_2) \setminus \sigma_1$, for all modules $\mathscr{P}$ and σ_1, σ_2.
4. $(\mathscr{P}_1 \cup \mathscr{P}_2) \setminus \sigma = (\mathscr{P}_1 \setminus \sigma) \cup (\mathscr{P}_2 \setminus \sigma)$, for all modules $\mathscr{P}_1$, $\mathscr{P}_2$ and signatures σ.
5. $(\overline{\mathscr{P}}^{\sigma_1}) \setminus \sigma_2 = \overline{\mathscr{P}}^{(\sigma_1 \setminus \sigma_2)}$, for all modules $\mathscr{P}$ and signatures σ_1, σ_2.
6. $\overline{\mathscr{P}}^{\sigma} = \overline{\mathscr{P}} \setminus (\sigma_e \setminus \sigma)$, for a module $\mathscr{P}$, with exported signature σ_e, and all σ.

Finally, we introduce a renaming operation that allows us to change function symbols with other function symbols of the same arity, in the global signature $\Sigma_\perp$. Therefore, given a module $\mathscr{P}$ and a function symbols renaming ρ, we define the renaming of $\mathscr{P}$ by ρ as a new module $\rho(\mathscr{P})$ where rules are conveniently renamed. The following definition formalizes this idea.

Definition 12 (*Renaming*)
Given a module $\mathscr{P} =< \sigma_p, \sigma_e, \mathscr{R} >$ and a function symbol renaming ρ, $\mathscr{P}$ renamed by ρ is the module $\rho(\mathscr{P}) =_{def}< \rho^*(\sigma_p) \setminus \rho^*(\sigma_e), \rho^*(\sigma_e), \rho^*(\mathscr{R}) >$, where $\rho^*(\sigma)$ is the signature resulting from applying ρ to all symbols in σ, and $\rho^*(\mathscr{R})$ is the set of rules resulting from applying ρ to all rules in $\mathscr{R}$.

The following example illustrates the usefulness of this operation to adequate parameter names of a module.

Example 13
In the module `OrdNatList\{isnat/1,leq/2}` of Example 11 we can rename the function symbol `isnat/1` with the new name `isbasetype/1` to obtain a more appropriate parameterized module

```
OrdList = {isnat/1 -> isbasetype/1}(OrdNatList\{isnat/1,leq/2}),
```

where we have denoted the corresponding renaming function ρ as the set of pairs $f/n \to \rho(f/n)$ such that $f/n \neq \rho(f/n)$. This module has the following appearance

```
OrdList =
  <{isbasetype/1,leq/2},
   {insert/2},
   {insert(X,[])     -> [X]               <= isbasetype(X) >< true.
    insert(X,[Y|Ys]) -> [X|[Y|Ys]]        <= leq(X,Y) >< true.
    insert(X,[Y|Ys]) -> [Y|insert(X,Ys)] <= leq(X,Y) >< false.} >
```

Now, the parameters become `isbasetype/2` and `leq/2`.

We will use this operation to change function names in exportation, importation and, specially, in instantiation for matching function names in the parameter signature of a module with function names in the exported signature of another module. See section 4.2 for some illustrative examples.

Proposition 8
Renaming of modules has the following properties, where ρ, ρ_1 and ρ_2 are function symbol renamings,

1. $\iota(\mathscr{P}) = \mathscr{P}$, for every module $\mathscr{P}$, where ι is the identity renaming.
2. $\rho(\mathscr{O}) = \mathscr{O}$, for every ρ.
3. $\rho_2(\rho_1(\mathscr{P})) = (\rho_2 \circ \rho_1)(\mathscr{P})$, for all modules $\mathscr{P}$ and all ρ_1, ρ_2.
4. $\rho(\mathscr{P}_1 \cup \mathscr{P}_2) = \rho(\mathscr{P}_1) \cup \rho(\mathscr{P}_2)$, for all modules $\mathscr{P}_1$, $\mathscr{P}_2$ and all ρ.
5. $\rho(\overline{\mathscr{P}}^{\sigma}) = \overline{\rho(\mathscr{P})}^{\rho^*(\sigma)}$, for all modules $\mathscr{P}$, signatures σ and injective ρ.
6. $\rho(\mathscr{P} \setminus \sigma) = \rho(\mathscr{P}) \setminus \rho^*(\sigma)$, for all modules $\mathscr{P}$, signatures σ and injective ρ.

4.2 Other modular constructions in CRWL-programming

Our notion of module is basically that of a program inside a context made up of other programs providing explicit rules for function symbols and implicit declarations of constructor symbols, all together defining a global signature $\Sigma_\perp$. In this section, we will show how the operations that we have defined above can be used to model typical module interconnections used in conventional modular programming languages. We will introduce new operations with modules for these relationships, but all these will be defined as derived expressions from the basic set. These expressions will reflect the relationship between the module denoted by the expression and its component modules, and the resulting modules will be interpreted as flat modules in all cases.

The closure of a module $\mathscr{M}$ w.r.t. a signature σ gives a form of encapsulation, hiding those function symbols in $\mathscr{M}$ that are not in σ, and making the function symbols in $\mathscr{M}$ and σ visible but only in an extensional way, i.e. by the results –

as partial constructor Σ-terms – of the function applications to constructor Σ-terms (including variables). Thus, we can provide an *export with encapsulation* operation '$\square$' over modules, in this simple way, $\sigma\square\mathscr{M} =_{def} \overline{\mathscr{M}}^{\sigma}$.

The union of modules reflects the behavior of some logic programming systems that allow adding new programs – saved in separate files – to the main database. With this operation, but modifying one of its arguments, we can express different forms of importation and instantiation. We can define an *import* operation $\ll$ between modules as the union of a module $\mathscr{M}$ – representing the body of the importing module – with the closure of the imported module $\mathscr{N}$ as $\mathscr{M} \ll \mathscr{N} =_{def} \mathscr{M} \cup \overline{\mathscr{N}}$. Module $\mathscr{M} \ll \mathscr{N}$ imports $\mathscr{N}$, which means that only the consequences of the functions defined in $\mathscr{N}$ are imported, and not their rules. When $exp(\mathscr{M}) \cap exp(\mathscr{N}) = \sigma_o$ we have a typical importation because functions defined in $\mathscr{N}$ are only reduced in $\mathscr{N}$. We can also express *selective importation* of a signature σ from $\mathscr{N}$ by combining importation with exportation to restrict the visible signature of the imported module, as $\mathscr{M} \ll (\sigma\square\mathscr{N})$, with $\sigma \subseteq exp(\mathscr{N})$. This expression is equivalent to $\mathscr{M} \cup \overline{\mathscr{N}}^{\sigma}$ by Proposition 6(*4*). *Multiple importation* or (selective) importation from several modules can be written as $(\ldots(\mathscr{M} \ll (\sigma_1\square\mathscr{N}_1))\ldots) \ll (\sigma_k\square\mathscr{N}_k)$, where the importation order is not relevant by Propositions 5(*3*,*4*) and 6(*4*,*5*). It can be easily proved that this expression is equivalent to the single importation $\mathscr{M} \ll ((\sigma_1\square\mathscr{N}_1)\cup\ldots\cup(\sigma_k\square\mathscr{N}_k))$. *Importation with renaming* can be expressed by an expression of the form $\mathscr{M} \ll \rho(\sigma\square\mathscr{N})$, with $\sigma \subseteq exp(\mathscr{N})$ and an injective function symbol renaming ρ (see Proposition 8(*5*)). By the properties of renaming this expression is equivalent to $\mathscr{M} \ll (\rho^*(\sigma)\square\rho(\mathscr{N}))$ and can be reduced to $\mathscr{M} \cup \rho(\overline{\mathscr{N}}^{\sigma})$.

Example 14
Let us consider the module `OrdList` in Example 13 and the new module

```
OrdNat =
  < {},
    {isnat/1, leq/2, geq/2},
    { isnat(zero)    -> true.
      isnat(succ(X)) -> isnat(X).
      leq(zero,zero)       -> true.
      leq(zero,succ(X))    -> isnat(X).
      leq(succ(X),zero)    -> false <= isnat(X) >< true.
      leq(succ(X),succ(Y)) -> leq(X,Y).
      geq(X,Y)             -> leq(Y,X). } >
```

where we define the predicate `isnat/1` and the two order relationships `leq/2` (less than or equal to) and `geq/2` (greater than or equal to). The importation

$$\texttt{OrdList} \ll \texttt{\{isnat/1 -> isbasetype/1\}(OrdNat)}$$

is a module with an infinite number of rules for `isbasetype/1`, `leq/2` and `geq/2` (all possible reductions to true or false), that behaves as calls to `isbasetype/1` and `leq/2` are reduced in `OrdNat` as calls to `isnat/1` and `leq/2` itself respectively.

Thus a typical program $\mathscr{M}$ with a hierarchical structure in the sense of standard modular programming, i.e., importing from several modules $\mathscr{N}_1,\ldots,\mathscr{N}_k$, possibly with renaming, can be built up from a plain program $\mathscr{P}$ – its body – and the imported modules as $\mathscr{M} = \mathscr{P} \ll (\rho_1(\sigma_1 \Box \mathscr{N}_1) \cup \ldots \cup \rho_k(\sigma_k \Box \mathscr{N}_k))$, with $\sigma_1 \subseteq exp(\mathscr{N}_1)$, $\ldots, \sigma_k \subseteq exp(\mathscr{N}_k)$ and $par(\mathscr{P}) \subseteq (\rho_1^*(\sigma_1) \cup \ldots \cup \rho_k^*(\sigma_k))$. This expression can be reduced to $\mathscr{P} \cup \rho_1(\overline{\mathscr{N}_1}^{\sigma_1}) \cup \ldots \cup \rho_k(\overline{\mathscr{N}_k}^{\sigma_k})$.

Because our basic modules can be parameterized, we can *instantiate* function symbols of the parameterized signature of a module $\mathscr{M}$ with function symbols, of the same arity but different name, exported by other module $\mathscr{N}$, simply by renaming suitably the parameters of $\mathscr{M}$ to fit (a part of) the exported signature of $\mathscr{N}$. Thus we obtain an *instantiation* operation that we denote $\mathscr{M}[\mathscr{N}, \rho]$ and define as $\mathscr{M}[\mathscr{N}, \rho] =_{def} \rho(\mathscr{M}) \ll \mathscr{N}$, where ρ is the function symbol renaming that characterizes the instantiation. This operation makes sense when $\rho^*(par(\mathscr{M})) \cap exp(\mathscr{N}) \neq \sigma_o$. When $par(\rho(\mathscr{M})) \subseteq exp(\mathscr{N})$ the instantiation is total and is partial in another case. Note that *instantiation* can be seen as a special form of importation. The difference between a (renamed) importation $\mathscr{M} \ll \rho(\mathscr{N})$ and an instantiation $\rho(\mathscr{M}) \ll \mathscr{N}$ is that in the former, symbols in the parameter signature of $\mathscr{M}$ refer to actual names in the exported signature of the imported module $\mathscr{N}$ (renamed by ρ), whereas in the latter, symbols in the parameter signature of $\mathscr{M}$ behave as true parameters being replaced by ρ with actual values of the exported signature of $\mathscr{N}$.

Example 15
Let us consider again the module `OrdList` in Example 13 and the module `OrdNat` defined in Example 14. The instantiation

```
OrdList[OrdNat,{isbasetype/1 -> isnat/1, leq/2 ->geq/2}]
```

is equivalent to a module, also with an infinite number of rules, but defining the predicates `isnat/1` and `geq/2` instead of `isbasetype/1` and `leq/2` , respectively.

Deletion of a signature σ in a module removes all rules defining function symbols in that signature but maintains the occurrences of these symbols in the rhs of the other rules. This operation can be used to *abstract a signature* σ from a module $\mathscr{M}$ in this way, $\mathscr{M}[\sigma] =_{def} \mathscr{M} \setminus \sigma$. This abstraction operation makes sense when $\sigma \subseteq exp(\mathscr{M})$ and each function symbol in σ appears in some rule of $rl(\mathscr{M} \setminus \sigma)$. This operation is very useful for making generic modules from concrete ones but unfortunately it is not implemented in conventional modular programming systems. As an example of the use of this operation we refer to Example 11. Also, with the deletion operation, we can model a sort of inheritance relationship between modules. *Inheritance with overriding* may be captured by means of union and deletion of a signature in this way, $\mathscr{M}\ isa\ \mathscr{N} =_{def} \mathscr{M} \cup (\mathscr{N} \setminus exp(\mathscr{M}))$. Module $\mathscr{M}\ isa\ \mathscr{N}$ inherits all functions in $\mathscr{N}$ – with their rules – not defined in $\mathscr{M}$ and uses the rules of $\mathscr{M}$ for all functions defined in $\mathscr{M}$, overriding the definition rules in $\mathscr{N}$, for common functions. In this case, overriding is carried out by deleting the common signature of the inherited module before adding it to the derived module.

Example 16
Let us consider a module defining some operations on polygonal lines and parameterized w.r.t. an addition operation `_+_/2`, a predicate `ispoint/1` to test if something

is a point, and operations `distance/2` and `translatepoint/2` for computing the distance between points and the point resulting of applying a translation, given by a vector (its second argument), to another point (its first argument).

```
Polygonal =
 <{_+_/2, ispoint/1, distance/2, translatepoint/2 },
  {perimeter/1,translate/2 },
  {perimeter([P1])         -> zero <= ispoint(P1) >< true.
   perimeter([P1|[P2|Ps]]) -> distance(P1,P2)+perimeter([P2|Ps]).
   translate([P1],V)         -> [translatepoint(P1,V)].
   translate([P1|[P2|Ps]],V) -> [translatepoint(P1,V)|translate([P2|Ps],V)].} >
```

(where we suppose that `distance/2` and `translatepoint/2` check that their arguments are points). Let us also consider another module defining some operations on squares and also parameterized w.r.t. a multiplication operation `_*_/2`, and the above operations `ispoint/1` and `distance/2`.

```
Square =
< {_*_/2, ispoint/1, distance/2},
  {issquare/1, side/1, perimeter/1, surface/1},
  {issquare([P1,P2,P3,P4]) -> true <= distance(P1,P2) >< distance(P2,P3),
                                      distance(P2,P3) >< distance(P3,P4),
                                      distance(P1,P2) >< distance(P3,P4).
    side([P1,P2,P3,P4]) -> distance(P1,P2) <= issquare([P1,P2,P3,P4]) >< true.
    perimeter(C) -> 4*side(C)       <= issquare(C) >< true.
    surface(C)   -> side(C)*side(C) <= issquare(C) >< true.} >.
```

With these modules we could define a new module `SquarePolygone` making module `Square` inherit from `Polygonal`,

```
SquarePolygone = Square isa Polygonal.
```

The resulting module would be

```
SquarePolygone =
< {_+_/2, _*_/2, ispoint/1, distance/2, translatepoint/2},
  {issquare/1, side/1, perimeter/1, surface/1, translate/2},
  {issquare([P1,P2,P3,P4]) -> true <= distance(P1,P2) >< distance(P2,P3),
                                      distance(P2,P3) >< distance(P3,P4),
                                      distance(P1,P2) >< distance(P3,P4).
    side([P1,P2,P3,P4]) -> distance(P1,P2) <= issquare([P1,P2,P3,P4]) >< true.
    perimeter(C) -> 4*side(C)       <= issquare(C) >< true.
    surface(C)   -> side(C)*side(C) <= issquare(C) >< true.
    translate([P1],V)         -> [translatepoint(P1,V)].
    translate([P1|[P2|Ps]],V) -> [translatepoint(P1,V)|translate([P2|Ps],V)].} >.
```

Note that `perimeter/1`, defined in the module `Polygonal`, has been redefined with the version of the module `Square`. The function `translate/2` has been inherited from `Polygonal`.

5 A compositional semantics

A module is basically a program because its interface can be extracted from its set of rules when we know the data constructor symbols, and operations defined on modules are operations on their sets of rules, i.e. operations on programs. The difference between a program and a program module is that a module can be thought of as a program piece that can be assembled with other pieces to build larger programs (this is one of the main reasons of making explicit their interfaces).

With this idea in mind, the model-theoretic semantics defined for CRWL-programs is not suitable for program modules because it is not compositional w.r.t. the operations defined over modules as we can see in the following example.

Example 17
Let Σ be a signature $(\{a/0, b/0, c/0\}, \{p/1, r/1\})$, and modules $\mathscr{P}_1$ and $\mathscr{P}_2$ with the following sets of rules: $rl(\mathscr{P}_1) = \{p(a) \to c\}$ and $rl(\mathscr{P}_2) = \{p(a) \to c,\, r(b) \to c \Leftarrow p(b) \bowtie c\}$. These modules have the same model-theoretic semantics, $\mathscr{M}_{\mathscr{P}_1} = \mathscr{M}_{\mathscr{P}_2}$, which is the CRWL-algebra $\mathscr{A}$ with functions $p^{\mathscr{A}}$ and $r^{\mathscr{A}}$ such that

$$\begin{array}{ll} p^{\mathscr{A}}(a) = \{c, \bot\}, & p^{\mathscr{A}}(b) = p^{\mathscr{A}}(c) = p^{\mathscr{A}}(\bot) = \{\bot\},\ p^{\mathscr{A}}(X) = \{\bot\},\ \forall X \in \mathscr{V} \\ r^{\mathscr{A}}(a) = \{\bot\}, & r^{\mathscr{A}}(b) = r^{\mathscr{A}}(c) = r^{\mathscr{A}}(\bot) = \{\bot\},\ r^{\mathscr{A}}(X) = \{\bot\},\ \forall X \in \mathscr{V}. \end{array}$$

However, their unions with $\mathscr{Q}$, such that $rl(\mathscr{Q}) = \{p(b) \to c\}$, have different model-theoretic semantics. The intended model of $\mathscr{P}_1 \cup \mathscr{Q}$ has a function $r^{\mathscr{M}_{\mathscr{P}_1 \cup \mathscr{Q}}}$ such that $r^{\mathscr{M}_{\mathscr{P}_1 \cup \mathscr{Q}}}(b) = \{\bot\}$, whereas $r^{\mathscr{M}_{\mathscr{P}_2 \cup \mathscr{Q}}}(b) = \{c, \bot\}$. So, $\mathscr{M}_{\mathscr{P}_1 \cup \mathscr{Q}} \neq \mathscr{M}_{\mathscr{P}_2 \cup \mathscr{Q}}$.

The compositionality of the semantics of a programming language is particularly relevant when modularity is involved. In fact, one of the most critical aspects in modular systems is the possibility of making a separate compilation of modules, and this can only be made in the presence of some kind of compositionality.

To study the compositionality and full abstraction of a semantics, we have to clearly set out these notions. We will adopt the approach proposed in Brogi and Turini (1995), where compositionality and full abstraction are defined in terms of the equivalence relation between programs induced by the semantics.

Definition 18 (*Compositional relation*)
Given an equivalence relation $\equiv$ defined between programs, an observation function Ob defined for programs, and a set $Oper$ of operations with programs, we say that

1. $\equiv$ *preserves* Ob iff for all programs $\mathscr{P}$ and $\mathscr{Q}$, $\mathscr{P} \equiv \mathscr{Q} \;\Rightarrow\; Ob(\mathscr{P}) = Ob(\mathscr{Q})$;
2. $\equiv$ is a *congruence* w.r.t. $Oper$ iff for all programs $\mathscr{P}_i$ and $\mathscr{Q}_i$ and all $O \in Oper$, $\mathscr{P}_i \equiv \mathscr{Q}_i$, for $i = 1, \ldots, n$, implies $O(\mathscr{P}_1, \ldots, \mathscr{P}_n) \equiv O(\mathscr{Q}_1, \ldots, \mathscr{Q}_n)$;
3. $\equiv$ is *compositional* w.r.t. $(Ob, Oper)$ iff it is a congruence w.r.t. $Oper$ and preserves Ob.

To set the notion of full abstraction for an equivalence relation, we need some way of distinguishing programs and for that reason we introduce the notion of *context*. Given a set of operations on programs $Oper$, and a metavariable $\mathscr{X}$, we define contexts $C[\![\mathscr{X}]\!]$ inductively as follows: $\mathscr{X}$ and each program is a context, also for each operation $O \in Oper$ with n program arguments and $C_1, \ldots, C_n$ contexts, $O(C_1, \ldots, C_n)$ is a context. Two programs $\mathscr{P}$ and $\mathscr{Q}$ are *distinguishable* under $(Ob, Oper)$ if there exists a context $C[\![\mathscr{X}]\!]$ such that $C[\![\mathscr{P}]\!]$ and $C[\![\mathscr{Q}]\!]$ have different external behavior, i.e. $Ob(C[\![\mathscr{P}]\!]) \neq Ob(C[\![\mathscr{Q}]\!])$. When $\mathscr{P}$ and $\mathscr{Q}$ are indistinguishable under $(Ob, Oper)$ we will write $\mathscr{P} \cong_{Ob,Oper} \mathscr{Q}$, i.e. for all contexts C, $Ob(C[\![\mathscr{P}]\!]) = Ob(C[\![\mathscr{Q}]\!])$.

Definition 19 (*Fully abstract relation*)
An equivalence relation $\equiv$ is *fully abstract* w.r.t. $(Ob, Oper)$ iff for all programs $\mathscr{P}$ and $\mathscr{Q}$, $\mathscr{P} \cong_{Ob,Oper} \mathscr{Q} \;\Rightarrow\; \mathscr{P} \equiv \mathscr{Q}$.

A semantics $\mathscr{S}$ for a programming language provides a meaning for programs and also induces an equivalence relation $\equiv_{\mathscr{S}}$ between programs: two programs are equivalent iff they have the same meaning in this semantics. This equivalence relation is used for defining compositionality and full abstraction for semantics.

Definition 20 (*Compositional and fully abstract semantics*)
A semantics $\mathscr{S}$ is compositional or fully abstract w.r.t. $(Ob, Oper)$ iff its corresponding relation $\equiv_{\mathscr{S}}$ is compositional or fully abstract, respectively, w.r.t. $(Ob, Oper)$.

Obviously, for each pair $(Ob, Oper)$ there exits a compositional and fully abstract relation between programs, the relation $\mathscr{P} \equiv_{(Ob,Oper)} \mathscr{Q}$ iff $Ob(C[\![\mathscr{P}]\!]) = Ob(C[\![\mathscr{Q}]\!])$, for every context $C[\![\mathscr{X}]\!]$. For each compositional equivalence relation $\equiv$, it is easy to see that $\mathscr{P} \equiv \mathscr{Q} \Rightarrow \mathscr{P} \equiv_{(Ob,Oper)} \mathscr{Q}$, and for each fully abstract equivalence relation $\equiv$, $\mathscr{P} \equiv_{(Ob,Oper)} \mathscr{Q} \Rightarrow \mathscr{P} \equiv \mathscr{Q}$. Thus, $\equiv_{(Ob,Oper)}$ will be the only equivalence relation which is both compositional and fully abstract w.r.t. $(Ob, Oper)$. And the more adequate semantics for programs (w.r.t. $(Ob, Oper)$) will be a semantics that induces this relation.

5.1 The $\mathscr{T}$-semantics

To find a compositional semantics we may think about programs as open in the sense that we can build up programs from other programs adding rules for new functions and also for already defined functions (of the signature Σ we were in) and imagine them as algebra transformers as is done in Mancarella and Pedreschi (1988) and Brogi (1993). The operator $\mathscr{T}_{\mathscr{P}}$ considered as a function $\mathbf{TAlg}_\Sigma \to \mathbf{TAlg}_\Sigma$ is a good candidate for the intended meaning of a program $\mathscr{P}$. First, we have to note that the set $[\mathbf{TAlg}_\Sigma \to \mathbf{TAlg}_\Sigma]$ of all continuous functions from $\mathbf{TAlg}_\Sigma$ to $\mathbf{TAlg}_\Sigma$, ordered by the relation $\mathscr{T}_1 \sqsubseteq \mathscr{T}_2$ iff $\mathscr{T}_1(\mathscr{A}) \sqsubseteq \mathscr{T}_2(\mathscr{A})$, for all $\mathscr{A} \in \mathbf{TAlg}_\Sigma$, with the least upper bound and the greatest lower bound of a finite set $\{\mathscr{T}_i\}_{i\in I}$ of functions pointwise defined as $(\sqcup_{i\in I}\mathscr{T}_i)(\mathscr{A}) = \sqcup_{i\in I}(\mathscr{T}_i(\mathscr{A}))$ and $(\sqcap_{i\in I}\mathscr{T}_i)(\mathscr{A}) = \sqcap_{i\in I}(\mathscr{T}_i(\mathscr{A}))$ respectively, and with bottom $\mathbb{T}_\perp$ and top $\mathbb{T}_\Sigma$ such that $\mathbb{T}_\perp(\mathscr{A}) = \perp_\Sigma$ and $\mathbb{T}_\Sigma(\mathscr{A}) = \top_\Sigma$, for all $\mathscr{A} \in \mathbf{TAlg}_\Sigma$, is a complete lattice as a consequence of $(\mathbf{TAlg}_\Sigma, \sqsubseteq)$ being a complete lattice. Now, we can associate a program with the corresponding immediate consequence operator, instead of its least fixpoint.

Definition 21 ($\mathscr{T}$-*semantics*)
We define the $\mathscr{T}$-semantics $\{\![\mathscr{P}]\!\}_T$ by denoting the meaning of a program $\mathscr{P}$ by its algebra transformer $\mathscr{T}_{\mathscr{P}}$, where $\mathscr{T}_{\mathscr{P}}$ is intended as $\mathscr{T}_{rl(\mathscr{P})}$.

This semantics entails the following equivalence relation on programs: $\mathscr{P} \equiv_T \mathscr{Q} \Leftrightarrow_{def} \mathscr{T}_{\mathscr{P}} = \mathscr{T}_{\mathscr{Q}}$. Thus, two programs are $\equiv_T$-equivalent if both define the same immediate consequences operator. In this context, and coinciding with logic programming, a natural choice of the observable behavior of a program $\mathscr{R}$ is its model-theoretic semantics. So we will adopt as observation function $Ob(\mathscr{R}) =_{def} \mathscr{M}_{\mathscr{R}}$. Notice that $\mathscr{M}_{\mathscr{R}}$ captures the graphs of all functions defined in $\mathscr{R}$, whereas functions not included in the program are considered totally undefined (their images only can be reduced to $\perp$). The semantics $\{\![\cdot]\!\}_T$ is compositional w.r.t. this observation function and the set of operations $Oper = \{\cup, \overline{(\cdot)}^{\sigma}, (\cdot)\backslash\sigma, \rho(\cdot)\}$. We can prove this fact by proving that $\{\![\cdot]\!\}_T$ is homomorphic in the following sense.

Theorem 22
Given a global signature Σ and a countable set of variable symbols $\mathscr{V}$, for all programs $\mathscr{P}$, $\mathscr{P}_1$ and $\mathscr{P}_2$ defined over Σ, every subsignature of function symbols $\sigma \subseteq FS_\Sigma$, and every function symbol renaming ρ, we have the following results

$$\begin{array}{lrcl} (a) & \{\![\mathscr{P}_1 \cup \mathscr{P}_2]\!\}_T & = & \{\![\mathscr{P}_1]\!\}_T \sqcup \{\![\mathscr{P}_2]\!\}_T\,; \\ (b) & \{\![\overline{\mathscr{P}}^{\sigma}]\!\}_T & = & \lambda\mathscr{A} \cdot (\{\![\mathscr{P}]\!\}_T^{\omega}(\perp_\Sigma))|_\sigma\,; \\ (c) & \{\![\mathscr{P} \setminus \sigma]\!\}_T & = & \{\![\mathscr{P}]\!\}_T \sqcap \mathbb{T}_{exp(\mathscr{P})\setminus\sigma}\,; \\ (d) & \{\![\rho(\mathscr{P})]\!\}_T & = & \mathscr{T}_{\rho^{-1}} \circ \{\![\mathscr{P}]\!\}_T \circ \mathscr{T}_\rho\,; \end{array}$$

where, for every algebra $\mathscr{A} \in \mathbf{TAlg}_\Sigma$ and every subsignature $\sigma \subseteq FS_\Sigma$, $\mathscr{A}|_\sigma$ is the term algebra characterized by $f^{\mathscr{A}|_\sigma}(\bar{t}) = f^{\mathscr{A}}(\bar{t})$, for $f/n \in \sigma$, and $f^{\mathscr{A}|_\sigma}(\bar{t}) = \{\perp\}$, otherwise. For each subsignature $\sigma \subseteq FS_\Sigma$, $\mathbb{T}_\sigma$ is the constant algebra transformer that, for all $\mathscr{A} \in \mathbf{TAlg}_\Sigma$ produces the same term algebra $\top_\sigma$ characterized by $f^{\top_\sigma}(\bar{t}) = \mathbf{CTerm}_\perp$, for $f/n \in \sigma$, and $f^{\top_\sigma}(\bar{t}) = \{\perp\}$, otherwise. And, for each rename ρ, $\mathscr{T}_\rho$ and $\mathscr{T}_{\rho^{-1}}$ are the algebra transformers defined by $\mathscr{T}_\rho(\mathscr{A}) = \mathscr{A}_\rho$ and $\mathscr{T}_{\rho^{-1}}(\mathscr{A}) = \mathscr{A}_{\rho^{-1}}$ where $\mathscr{A}_\rho$ and $\mathscr{A}_{\rho^{-1}}$ are the term algebras characterized by

$$f^{\mathscr{A}_\rho} = \rho(f)^{\mathscr{A}} \quad \text{and} \quad f^{\mathscr{A}_{\rho^{-1}}} = \begin{cases} \sqcup\{g^{\mathscr{A}} \mid f = \rho(g)\}, & \text{when this set is not empty,} \\ f^{\perp_\Sigma} & \text{otherwise,} \end{cases}$$

for every function symbol f in FS_Σ.

Thus, the meaning of the union of two programs (a) can be extracted from the meaning of each one, the meaning of the closure of a program (b) is obtained from the fixpoint of the program semantics, and deleting a signature from a program (c) is semantically equivalent to the intersection of the program semantics with an algebra transformer which depends on the exported signature of the program. Nevertheless, the intersection we are mentioning here is not an operation over programs (as in Brogi (1993)) but an operation on algebra transformers. The meaning of a renamed program (d) can be obtained as the composition of the meaning of the program with two algebra transformers associated with the renaming and its reverse.

Corollary 1 (*Compositionality of* $\{\![\cdot]\!\}_T$)
The semantics $\{\![\cdot]\!\}_T$ is compositional with respect to $(Ob, \{\cup, \overline{(\cdot)}^{\sigma}, (\cdot)\setminus\sigma, \rho(\cdot)\})$.

As the above corollary states, $\{\![\cdot]\!\}_T$ is compositional w.r.t. union, closure, deletion and renaming, when the canonic model of a program is taken as its observable behavior. However, the following example shows that it is not fully abstract.

Example 23
Let Σ be a signature $(\{c/0, d/0\}, \{f/0\})$ and let $\mathscr{P}$ and $\mathscr{Q}$ be the modules such that $rl(\mathscr{P}) = \{f \to c,\ f \to d\}$ and $rl(\mathscr{Q}) = \{f \to c,\ f \to d \Leftarrow f \bowtie c\}$. They are indistinguishable under $\{\cup, \overline{(\cdot)}^{\sigma}, (\cdot)\setminus\sigma, \rho(\cdot)\}$, but they are not $\equiv_T$-equivalent. In fact, $\mathscr{T}_{\mathscr{P}}(\perp_\Sigma) \neq \mathscr{T}_{\mathscr{Q}}(\perp_\Sigma)$ because $f^{\mathscr{T}_{\mathscr{P}}(\perp_\Sigma)} = \{c, d, \perp\}$ whereas $f^{\mathscr{T}_{\mathscr{Q}}(\perp_\Sigma)} = \{c, \perp\}$.

The $\mathscr{T}$-semantics distinguishes more than the model-theoretic semantics, since the immediate consequence operator captures what is happening in each reduction step, but the non-full abstraction result means that this semantics distinguishes more than necessary. It is too fine. In the next section we will try a coarser semantics – also studied in logic programming (Brogi and Turini, 1995) – defined from the sets of pre-fixpoints of $\mathscr{T}$.

6 A fully abstract semantics

In this section, a fully abstract semantics is presented, which is also compositional except for the deletion operation. For a better motivation, we will not introduce this semantics directly. Instead, we will define a first approximation, the so-called *term model semantics* (Definition 24), which only is compositional (w.r.t. the union, closure and renaming operations), and then we will obtain the full abstraction property by restricting the term models (Definition 29).

6.1 The term model semantics

Formally, we will introduce the first semantics by directly considering the corresponding equivalence relation.

Definition 24 (*Model equivalence*)
Two programs $\mathscr{P}$ and $\mathscr{Q}$ are model-equivalent, $\mathscr{P} \equiv_M \mathscr{Q}$, iff their algebra transformers have the same pre-fixpoints.

By Lemma 3 this means that two programs are equivalent iff they have the same term models. This equivalence relation corresponds to the following semantics:

$$\{\![\mathscr{P}]\!\}_M =_{def} \{\mathscr{M} \mid \mathscr{M} \text{ is a term model of } \mathscr{P}\}$$

which will be called *loose model-theoretic semantics*, or simply *term model semantics*. To derive the corresponding result about compositionality, we need an auxiliary property about $\mathscr{T}_\rho$ and $\mathscr{T}_{\rho^{-1}}$.

Lemma 5
Given two term algebras $\mathscr{A}, \mathscr{B} \in \mathbf{TAlg}_\Sigma$, for every function symbol renaming ρ,

$$\mathscr{A}_{\rho^{-1}} \sqsubseteq \mathscr{B} \Leftrightarrow \mathscr{A} \sqsubseteq \mathscr{B}_\rho \text{ or, equivalently, } \mathscr{T}_{\rho^{-1}}(\mathscr{A}) \sqsubseteq \mathscr{B} \Leftrightarrow \mathscr{A} \sqsubseteq \mathscr{T}_\rho(\mathscr{B}).$$

This lemma claims that $\mathscr{T}_{\rho^{-1}}$ is, essentially, the reverse operator for $\mathscr{T}_\rho$.

Theorem 25 (*Compositionality of* $\{\![\cdot]\!\}_M$)
For all programs $\mathscr{P}, \mathscr{Q}, \mathscr{P}_i, \mathscr{Q}_i$,

1. $\mathscr{P} \equiv_M \mathscr{Q}$ implies $Ob(\mathscr{P}) = Ob(\mathscr{Q})$.
2. $\mathscr{P}_i \equiv_M \mathscr{Q}_i$ for $i = 1, 2$, implies $\mathscr{P}_1 \cup \mathscr{P}_2 \equiv_M \mathscr{Q}_1 \cup \mathscr{Q}_2$.
3. $\mathscr{P} \equiv_M \mathscr{Q}$ implies $\overline{\mathscr{P}}^\sigma \equiv_M \overline{\mathscr{Q}}^\sigma$, for every signature σ.
4. $\mathscr{P} \equiv_M \mathscr{Q}$ implies $\rho(\mathscr{P}) \equiv_M \rho(\mathscr{Q})$, for every function symbol renaming ρ.

Therefore, the semantics $\{\![\cdot]\!\}_M$ is compositional w.r.t. $(Ob, \{\cup, \overline{(\cdot)}^\sigma, \rho(\cdot)\})$.

Unfortunately, this semantics is not compositional w.r.t. deletion.

Example 26
Let Σ be the signature $(\{a/0, b/0\}, \{f/0, g/0\})$ and let $\mathscr{P}$ and $\mathscr{Q}$ be two modules with rules $rl(\mathscr{P}) = \{f \to a,\ g \to b\}$ and $rl(\mathscr{Q}) = \{f \to a,\ g \to b \Leftarrow f \bowtie a\}$. Both modules have the same term models, those term algebras $\mathscr{A}$ with $a \in f^{\mathscr{A}}$ and $b \in g^{\mathscr{A}}$. But by deleting $f/0$ in each module we have $\mathscr{P} \setminus \{f/0\}$ and $\mathscr{Q} \setminus \{f/0\}$ with $rl(\mathscr{P} \setminus \{f/0\}) = \{g \to b\}$ and $rl(\mathscr{Q} \setminus \{f/0\}) = \{g \to b \Leftarrow f \bowtie a\}$, and now $\perp_\Sigma$ is a model of $\mathscr{Q} \setminus \{f/0\}$ whereas it is not a model of $\mathscr{P} \setminus \{f/0\}$.

For a different reason, the semantics $\{\![\cdot]\!\}_M$ is not fully abstract.

Example 27
Let Σ be the signature $(\{a/0\}, \{f/0, g/1\})$ and let $\mathscr{P}$ and $\mathscr{Q}$ be two modules with rules $rl(\mathscr{P}) = \{f \to a \Leftarrow g(a) \bowtie a\}$ and $rl(\mathscr{Q}) = \{f \to a \Leftarrow g(X) \bowtie a\}$, where the rule in $\mathscr{P}$ is an instance of the rule in $\mathscr{Q}$. Obviously, both modules are indistinguishable but they do not have the same term models. In fact, if we consider the algebra $\mathscr{A}$ such that: $f^{\mathscr{A}} = \{\perp\}$, $g^{\mathscr{A}}(X) = \{a, \perp\}$ and $g^{\mathscr{A}}(a) = \{\perp\}$, $\mathscr{A}$ is a model of $\mathscr{P}$ but it is not a model of $\mathscr{Q}$.

6.2 The consistent term model semantics

To prove the full abstraction property we need to consider a different equivalence relation (i.e. semantics). If we observe the above counter-example, we can see that, for the term algebra $\mathscr{A}$ used to distinguish $\{\![\mathscr{P}]\!\}_M$ from $\{\![\mathscr{Q}]\!\}_M$, $g^{\mathscr{A}}(X) = \{a, \perp\}$ and $g^{\mathscr{A}}(a) = \{\perp\}$; that is, $\mathscr{A}$ is such that the instantiation of the variable X derives in a loss of information for the interpretation of g because $g^{\mathscr{A}}(X\theta)$ is smaller than $(g^{\mathscr{A}}(X))\theta$, for $\theta = \{X/a\}$. In general, the notion of term algebra (see section 2) does not impose any relation between $g^{\mathscr{A}}(\bar{t}\theta)$ and $(g^{\mathscr{A}}(\bar{t}))\theta$. This is not reasonable if we take into account the role of term algebras when they are used to model programs. On the contrary, the interpretation of a function symbol (in a term algebra) applied to arguments with variables must be related to the interpretation of the same function symbol when these variables are instantiated. With this idea in mind, we introduce the notion of *consistency* in a term algebra.

Definition 28 (*Consistency of term algebras*)
A term algebra $\mathscr{A} \in \mathbf{TAlg}_\Sigma$ is consistent iff for every $f \in FS^n_\Sigma$ and $t_i \in \mathbf{CTerm}_\perp$ $(i = 1, \ldots, n)$, $f^{\mathscr{A}}(\bar{t}\theta) \supseteq (f^{\mathscr{A}}(\bar{t}))\theta$ for all $\theta \in \mathbf{CSubst}$, where $(f^{\mathscr{A}}(\bar{t}))\theta$ stands for the set $\{u\theta \mid u \in f^{\mathscr{A}}(\bar{t})\}$.

We denote by $\mathbf{CTAlg}_\Sigma$ the family of all consistent term algebras. Note that consistency is only required for total substitutions (i.e. substitutions which do not include partial constructor terms). This is due to the special treatment of $\perp$, which is considered as lack of information. The notion of consistency here introduced is close to that of *closure under substitutions* defined for interpretations in Apt (1996), and is also related with the notion of C-interpretation considered in Falaschi *et al.* (1993), but our requirements are weaker than those. To justify the reasonable nature of consistent term algebras we will prove several desirable properties. For instance, the immediate consequences operator maps consistent algebras into consistent algebras, and the canonical model of a program is consistent.

Lemma 6
For every $\mathscr{A} \in \mathbf{CTAlg}_\Sigma$, $r \in \mathbf{Term}_\perp$, and $\theta \in \mathbf{CSubst}$, $[\![r]\!]^{\mathscr{A}}_{id}\theta \subseteq [\![r\theta]\!]^{\mathscr{A}}_{id}$.

Proposition 9
Given a program $\mathscr{P}$, if $\mathscr{A} \in \mathbf{CTAlg}_\Sigma$, then $\mathscr{T}_{\mathscr{P}}(\mathscr{A}) \in \mathbf{CTAlg}_\Sigma$.

Proposition 10
Given a program $\mathscr{P}$, the canonical term model $\mathscr{M}_{\mathscr{P}}$ is consistent.

Now, we may define an equivalence relation based only on consistent term models.

Definition 29 (*Consistent model equivalence*)
Two programs $\mathscr{P}$ and $\mathscr{Q}$ are consistent model-equivalent, $\mathscr{P} \equiv_{CM} \mathscr{Q}$, iff their have the same consistent models.

This equivalence is clearly weaker than the model equivalence and corresponds to the following semantics:

$$\{\![\mathscr{P}]\!\}_{CM} =_{def} \{\mathscr{M} \mid \mathscr{M} \text{ is a consistent term model of } \mathscr{P}\}$$

which will be called *loose consistent model-theoretic semantics*, or simply *consistent term model semantics*. Obviously, $\{\![\mathscr{P}]\!\}_{CM} = \{\![\mathscr{P}]\!\}_M \cap \mathbf{CTAlg}_\Sigma$, and the compositionality property of this semantics may be obtained in a similar way as the compositionality of the term model semantics.

Theorem 26 (*Compositionality of* $\{\![\cdot]\!\}_{CM}$)
For all programs $\mathscr{P}, \mathscr{Q}, \mathscr{P}_i, \mathscr{Q}_i$,

1. $\mathscr{P} \equiv_{CM} \mathscr{Q}$ implies $Ob(\mathscr{P}) = Ob(\mathscr{Q})$.
2. $\mathscr{P}_i \equiv_{CM} \mathscr{Q}_i$ for $i = 1, 2$, implies $\mathscr{P}_1 \cup \mathscr{P}_2 \equiv_{CM} \mathscr{Q}_1 \cup \mathscr{Q}_2$.
3. $\mathscr{P} \equiv_{CM} \mathscr{Q}$ implies $\overline{\mathscr{P}}^\sigma \equiv_{CM} \overline{\mathscr{Q}}^\sigma$, for every signature σ.
4. $\mathscr{P} \equiv_{CM} \mathscr{Q}$ implies $\rho(\mathscr{P}) \equiv_{CM} \rho(\mathscr{Q})$, for every function symbol renaming ρ.

Therefore, the semantics $\{\![\cdot]\!\}_{CM}$ is compositional w.r.t. $(Ob, \{\cup, \overline{(\cdot)}^\sigma, \rho(\cdot)\})$.

Example 26 also illustrates the non-compositionality of $\{\![\cdot]\!\}_{CM}$ w.r.t. the deletion operation because the programs $\mathscr{P}$ and $\mathscr{Q}$ only define functions without arguments. However, this semantics is fully abstract; to prove this fact, we need an auxiliary result, showing how a (*minimal*) program $\mathscr{P}$ can be constructed from a consistent term algebra $\mathscr{A}$ and an element $t \in [\![r]\!]^{\mathscr{A}}_{id}$ such that $\mathscr{A}$ is a model of $\mathscr{P}$ and $t \in [\![r]\!]^{\mathscr{M}_{\mathscr{P}}}_{id}$. Proposition 11 formalizes this idea. In order to simplify the proof of this result, we will prove some properties about the notion of *canonical rewrite rule* already introduced in Definition 6.

Lemma 7
For each canonical rewrite rule $crr(e, r)$, $\mathscr{T}_{\{crr(e,r)\}}$ is constant and if $e = f(\bar{t})$ then, for every term algebra $\mathscr{A}$,

$$h^{\mathscr{T}_{\{crr(e,r)\}}(\mathscr{A})}(\bar{s}) = \begin{cases} \bigcup_{\eta \in \mathbf{CSubst}} \{ [\![r\eta]\!]^{\mathscr{A}}_{id} \mid \bar{t}\eta \sqsubseteq \bar{s}\} \cup \{\perp\} & \text{if } h = f, \\ \{\perp\} & \text{otherwise} \end{cases}$$

Proposition 11
Let $\mathscr{A} \in \mathbf{CTAlg}_\Sigma$ be a consistent term algebra, and $r \in \mathbf{Term}_\perp$. Then, for every $t \in [\![r]\!]^{\mathscr{A}}_{id}$, a program $\mathscr{R}_t$ exists such that $t \in [\![r]\!]^{\mathscr{M}_{\mathscr{R}_t}}_{id}$ and $\mathscr{T}_{\mathscr{R}_t}(\mathscr{A}) \sqsubseteq \mathscr{A}$. Moreover, $\mathscr{T}_{\mathscr{R}_t}$ is constant.

Now, we can obtain the full abstraction property for $\{\![\cdot]\!\}_{CM}$.

Theorem 31 (*Full abstraction of* $\{\![\cdot]\!\}_{CM}$)
The semantics $\{\![\cdot]\!\}_{CM}$ is fully abstract w.r.t. $(Ob,\ \{\cup, \overline{(\cdot)}^{\sigma}, (\cdot)\backslash\sigma, \rho(\cdot)\})$

7 A compositional and fully abstract semantics

The fact that the consistent term model semantics is fully abstract but not compositional w.r.t. the deletion of a subsignature means that this semantics is more abstract than necessary. We need a finer semantics but not as fine as the $\mathscr{T}$-semantics. One way of obtaining such a semantics is by increasing the number of pre-fixpoints (related to the $\mathscr{T}$-operator) to be considered when we compare two programs, and to obtain compositionality w.r.t. the deletion operation, we may consider the consistent term models of all programs obtained by deleting a subsignature. With this idea we define the following equivalence between programs

Definition 32 (*Deletion equivalence*)
For programs $\mathscr{P}$ and $\mathscr{Q}$, we define the deletion equivalence $\mathscr{P} \equiv_D \mathscr{Q}$ as $\mathscr{P}\backslash\sigma \equiv_{CM} \mathscr{Q}\backslash\sigma$ for all subsignatures $\sigma \subseteq FS_\Sigma$.

This equivalence is finer than the consistent model equivalence and coarser than the equivalence induced by the $\mathscr{T}$-semantics. In fact, $\mathscr{P} \equiv_D \mathscr{Q}$ implies $\mathscr{P} \equiv_{CM} \mathscr{Q}$ because this relationship coincides with $\mathscr{P} \setminus \sigma_0 \equiv_{CM} \mathscr{Q} \setminus \sigma_0$, where σ_0 is the empty signature. And if $\mathscr{P} \equiv_T \mathscr{Q}$, or equivalently $\mathscr{T}_{\mathscr{P}} = \mathscr{T}_{\mathscr{Q}}$, it can be proved that $\mathscr{T}_{\mathscr{P}\backslash\sigma} = \mathscr{T}_{\mathscr{Q}\backslash\sigma}$, for all $\sigma \subseteq FS_\Sigma$, and then $\mathscr{P} \setminus \sigma \equiv_{CM} \mathscr{Q} \setminus \sigma$, for all $\sigma \subseteq FS_\Sigma$, which is $\mathscr{P} \equiv_D \mathscr{Q}$. The deletion equivalence is compositional w.r.t. all operations.

Theorem 33 (*Compositionality of* $\equiv_D$)
For all programs $\mathscr{P}, \mathscr{Q}, \mathscr{P}_i, \mathscr{Q}_i$,

1. $\mathscr{P} \equiv_D \mathscr{Q}$ implies $Ob(\mathscr{P}) = Ob(\mathscr{Q})$.
2. $\mathscr{P}_i \equiv_D \mathscr{Q}_i$ for $i = 1, 2$, implies $\mathscr{P}_1 \cup \mathscr{P}_2 \equiv_D \mathscr{Q}_1 \cup \mathscr{Q}_2$.
3. $\mathscr{P} \equiv_D \mathscr{Q}$ implies $\overline{\mathscr{P}}^{\sigma} \equiv_D \overline{\mathscr{Q}}^{\sigma}$, for every signature $\sigma \subseteq FS_\Sigma$.
4. $\mathscr{P} \equiv_D \mathscr{Q}$ implies $\mathscr{P} \setminus \sigma \equiv_D \mathscr{Q} \setminus \sigma$, for every signature $\sigma \subseteq FS_\Sigma$.
5. $\mathscr{P} \equiv_D \mathscr{Q}$ implies $\rho(\mathscr{P}) \equiv_D \rho(\mathscr{Q})$, for every function symbol renaming ρ.

Thus, the equivalence $\equiv_D$ is compositional w.r.t. $(Ob, \{\cup, \overline{(\cdot)}^{\sigma}, (\cdot) \setminus \sigma, \rho(\cdot)\})$.

Theorem 34 (*Full abstraction of* $\equiv_D$)
The equivalence $\equiv_D$ is fully abstract w.r.t. $(Ob,\ \{\cup, \overline{(\cdot)}^{\sigma}, (\cdot)\backslash\sigma, \rho(\cdot)\})$

Definition 35 (*Deletion semantics*)
We define the deletion semantics $\{\![\mathscr{P}]\!\}_D$, of a program $\mathscr{P}$, as $\{\mathbf{M}_{f/n}(\mathscr{P}) \mid f/n \in FS_\Sigma\}$, where $\mathbf{M}_{f/n}(\mathscr{P})$ is the set of all consistent term models of the rules of $\mathscr{P}$ that define f/n.

The deletion semantics induces the deletion equivalence.

Proposition 12

$$\mathscr{P} \equiv_D \mathscr{Q} \Leftrightarrow_{def} \{\![\mathscr{P}]\!\}_D = \{\![\mathscr{Q}]\!\}_D$$

Thus, the deletion semantics is compositional and fully abstract w.r.t. $(Ob, \{\cup, \overline{(\cdot)}^{\sigma}, (\cdot) \setminus \sigma, \rho(\cdot)\})$.

8 Introducing hidden symbols

In this section we explore an alternative to modules with an infinite number of rules, generated by the closure operation, that also supports local constructor symbols. For this aim we will consider a global or *visible* signature Σ and a set $\mathscr{V}$ of variable symbols together with a new set Ω of labels that we identify with the set of module names and module expressions. With this set we obtain a *labeled signature* $\Omega \times \Sigma = (\Omega \times DS_\Sigma, \Omega \times FS_\Sigma)$ which we will consider as protected or non accessible for users and writers of modules, i.e. *hidden*. This signature will be only managed by the module system for internal representation of module expressions. Pairs (M, f) of $\Omega \times \Sigma$, called *labeled* symbols, will be denoted by $M.f$.

As we have seen in section 4.1 the purpose of the closure of a module is to hide the definitions of function symbols, making only their results visible. To this aim, the rules of a module are replaced with all (possibly infinite) approximations that can be derived from them. However, we can obtain an *internal representation* of the closure operation, with a finite number of rules, with the aid of labeled symbols, following an idea that appears in Brogi (1993) applied to the hiding of predicate definitions in logic programs. We go further into this idea applying it to deal with local constructor symbols.

8.1 A finite representation of closure

Let $\mathscr{P} =< \sigma_p, \sigma_e, \mathscr{R} >$ a module of $\mathbf{PMod}(\Sigma_\perp)$ with a finite set of rules. We can protect its rules translating them to a protected signature by labeling all function symbols with the module's name and introducing a bridge rule $f(\overline{X}) \to \mathscr{P}.f(\overline{X})$ for each function symbol $f/n \in \sigma_e$. In this way we obtain a module $\mathscr{P}^*$ in the signature $\overline{\Sigma}_\perp = (DS_{\Sigma_\perp}, FS_{\Sigma_\perp} \cup (\Omega \times FS_{\Sigma_\perp}))$ with an isolated (hidden) part $\mathscr{R}_H$, made up of all translated rules, and a bridge part $\mathscr{R}_B$ for accessing the isolated part, made up of all bridge rules. Obviously with this module we can derive the same approximations, for visible function symbols, as with $\overline{\mathscr{P}}$ in every context. We will call these modules *structured modules* to distinguish them from *plain modules* used up to now. In general, a structured module will be a module $< \sigma_p, \sigma_e, \mathscr{R}_V \cup \mathscr{R}_B \cup \mathscr{R}_H >$ with a visible parameter signature σ_p, a visible exported signature σ_e, and a set of rules with three – possibly empty – parts, a visible part $\mathscr{R}_V$ made up of rules only with function symbols in FS_Σ, a hidden part $\mathscr{R}_H$ made up of rules only with function symbols in $\Omega \times FS_\Sigma$, and a bridge part $\mathscr{R}_B$ made up of bridge rules $f(\overline{X}) \to \mathscr{P}.g(\overline{X})$, for any label $\mathscr{P} \in \Omega$, such that each symbol $\mathscr{P}.g$ has a definition rule in $\mathscr{R}_H$. Also, σ_e is made up of all function symbols with a definition rule in $\mathscr{R}_V$ or $\mathscr{R}_B$, and σ_p is made up of all parameter function symbols which appear in $\mathscr{R}_V$. We define union, deletion of functional signature and renaming in the same way as we did in section 4.1, but we will use deletion and renaming involving only visible signature, and, instead of closure, we define a *structured closure* for a structured module $\mathscr{P} =< \sigma_p, \sigma_e, \mathscr{R}_V \cup \mathscr{R}_B \cup \mathscr{R}_H >$ as the module $\mathscr{P}^* =< \emptyset, \sigma_e, \mathscr{R}_B^* \cup \mathscr{R}_H^* >$ obtained by applying the renaming $\tau(\mathscr{P})$, that transforms each visible function symbol f of $\mathscr{R}_V$ and $\mathscr{R}_B$ into $\mathscr{P}.f$ and maintains all labeled symbols, and adding new bridge rules corresponding to the function symbols of σ_e. Now, we can define a representation morphism from modular expressions made up from finite plain modules to structured modules in the following way:

- $\imath(\mathscr{P}) = \mathscr{P}$, for each finite plain module $\mathscr{P}$;
- $\imath(\mathscr{P} \cup \mathscr{Q}) = \imath(\mathscr{P}) \cup \imath(\mathscr{Q})$, for module expressions $\mathscr{P}$ and $\mathscr{Q}$;
- $\imath(\mathscr{P} \setminus \sigma) = \imath(\mathscr{P}) \setminus \sigma$, for each module expression $\mathscr{P}$ and visible signature σ;
- $\imath(\rho(\mathscr{P})) = \rho(\imath(\mathscr{P}))$, for each module expression $\mathscr{P}$ and visible signature renaming ρ;
- $\imath(\overline{\mathscr{P}}) = (\imath(\mathscr{P}))^*$, for each module expression $\mathscr{P}$.

Example 36
Let `OrdList` and `OrdNat` be the modules defined in Examples 13 and 14, respectively, and let `P` be the name of the module $\imath$(`OrdNat`). The representation of `OrdList` $\cup$ `{isnat/1->isbasetype/1}`($\overline{\texttt{OrdNat}}$) will be the structured module $\imath$(`OrdList`) $\cup$ `{isnat/1->isbasetype/1}`($\texttt{P}^*$), with the following aspect

```
<{}, {isbasetype/1,leq/2,geq/2,insert/2},
 {   % visible rules
  insert(X,[])     -> [X]              <= isbasetype(X) >< true.
  insert(X,[Y|Ys]) -> [X|[Y|Ys]]       <= leq(X,Y) >< true.
  insert(X,[Y|Ys]) -> [Y|insert(X,Ys)] <= leq(X,Y) >< false.
     % bridge rules
  isbasetype(X) -> P.isnat(X).
  leq(X,Y) -> P.leq(X,Y).
  geq(X,Y) -> P.geq(X,Y).
     % hidden rules
  P.isnat(zero)     -> true.
  P.isnat(succ(X)) -> P.isnat(X).
  P.leq(zero,zero)       -> true.
  P.leq(zero,succ(X))    -> P.isnat(X).
  P.leq(succ(X),zero)    -> false <= P.isnat(X) >< true.
  P.leq(succ(X),succ(Y)) -> P.leq(X,Y).
  P.geq(X,Y)             -> P.leq(Y,X). } >
```

The behaviour of a structured module $\mathscr{P} =< \sigma_p, \sigma_e, \mathscr{R}_V \cup \mathscr{R}_B \cup \mathscr{R}_H >$ w.r.t. the visible signature can be expressed with the aid of the algebra transformer $\mathscr{U}_{\mathscr{P}} : \mathbf{CTAlg}_\Sigma \rightarrow \mathbf{CTAlg}_\Sigma$ defined, for each $\mathscr{A}$, as $\mathscr{U}_{\mathscr{P}}(\mathscr{A}) = \mathscr{T}_{\mathscr{R}_V \cup \mathscr{R}_B}(\mathscr{T}^{\omega}_{\mathscr{R}_H}(\bot_{\overline{\Sigma}}) \sqcup \overline{\mathscr{A}})|_\Sigma$, where $\overline{\mathscr{A}}$ is the extension of $\mathscr{A}$ to an algebra of $\mathbf{CTAlg}_{\overline{\Sigma}}$ obtained by adding functions $\mathscr{P}.f^{\overline{\mathscr{A}}}$ defined as $\mathscr{P}.f^{\overline{\mathscr{A}}}(\bar{t}) = \langle \bot \rangle$, for each $f/n \in FS_\Sigma$ and $\mathscr{P} \in \Omega$, and $\mathscr{B}|_\Sigma$ means the reduct of the algebra $\mathscr{B} \in \mathbf{CTAlg}_{\overline{\Sigma}}$ obtained by forgetting all functions denoting labeled function symbols. In this expression, $\mathscr{T}^{\omega}_{\mathscr{R}_H}(\bot_{\overline{\Sigma}})$ represents all the information which can be obtained from the hidden rules; this information is added to the extended algebra because it has to be available for the immediate consequences operator corresponding to the visible and bridge rules, to obtain the approximations for visible functions. The relationship, at the semantical level, between programs and structured modules is given in the following theorem.

Theorem 37
For each modular expression $\mathscr{E}$, made up from finite plain programs, and its implementation $\imath(\mathscr{E})$, we have $\mathscr{T}_{\mathscr{E}} = \mathscr{U}_{\imath(\mathscr{E})}$.

From this theorem we obtain that for two equivalent module expressions $\mathscr{P}$ and $\mathscr{Q}$ (i.e. $\mathscr{P}$ and $\mathscr{Q}$ have the same components but, possibly, different expressions with the operations), $\mathscr{U}_{\imath(\mathscr{P})} = \mathscr{U}_{\imath(\mathscr{Q})}$ although it is possible that $\imath(\mathscr{P})$ differs from $\imath(\mathscr{Q})$ due to the occurrence of closure operations. Also, the models of a program module $\mathscr{P}$ will be the pre-fixpoints of $\mathscr{U}_{\imath(\mathscr{P})}$ and we can define the visible semantics of structured modules based on this operator. In particular we obtain the deletion semantics by considering, for each structured module $\mathscr{P} =< \sigma_p, \sigma_e, \mathscr{R}_V \cup \mathscr{R}_B \cup \mathscr{R}_H >$, the indexed family of sets of pre-fixpoints of $\mathscr{U}_{\mathscr{P}\backslash(\sigma_e \backslash f)}$ for each $f/n \in \sigma_e$.

8.2 Local constructor symbols

To simplify the theoretical study of programs composition in CRWL-programming, and to capture the idea of module as open program, we have assumed that constructor symbols are common to all programs. However, as it was discussed in section 4, this assumption prevents to hide constructor symbols, what is not acceptable from a practical point of view.

We can hide constructor symbols by labeling them as we have done with function symbols to protect them against user manipulations. Labeled constructor symbols can only be manipulated in the internal representation of the closure of the module corresponding to their label. Outside this module, function symbols defined on labeled constructor symbols can only be applied to variable symbols or to other function applications that can be reduced to this labeled constructor symbols. To realize this idea we only need to modify our closure implementation extending it to manage constructor symbols also. So, we define *closure hiding a subsignature C of constructor symbols* for a module $\mathscr{P}$ as a (non plain) module $\overline{\mathscr{P}}_C$ such that $\imath(\overline{\mathscr{P}}_C) = \mathscr{P}^*_C$ where $\mathscr{P}^*_C$ is obtained as $\mathscr{P}^*$ but now the renaming $\tau(\mathscr{P})$ also transforms each visible constructor symbol c of C into $\mathscr{P}.c$.

Example 38
Let us suppose a module `LNat` for lists of natural numbers exporting the function symbols `isnat/1`, `_<_/2` and `_++_/2`, and consider the following module for binary search trees where tree constructors `nil/0` and `mktree/3` are used.

```
BST =
<{isnat/1, _<_/2, _++_/2}, {empty/0, insert/2, inorder/1},
 {empty -> nil .
  insert(N,nil) -> mktree(N,nil,nil) <= isnat(N) >< true .
  insert(N,mktree(M,T1,T2)) -> mktree(M,T1,T2) <= N >< M, isnat(N) >< true .
  insert(N,mktree(M,T1,T2)) -> mktree(M,insert(N,T1),T2) <= N<M >< true .
  insert(N,mktree(M,T1,T2)) -> mktree(M,T1,insert(N,T2)) <= M<N >< true .
  inorder(nil) -> [] .
  inorder(mktree(M,T1,T2)) -> inorder(T1)++[M|inorder(T2)] <= isnat(M) >< true .}>
```

We may hide the tree constructors by considering $\overline{(\texttt{LNat} \cup \texttt{BST})}_{\{nil,mktree\}}$. This module will have the following representation:

```
<{}, {isnat/1, _<_/2, _++_/2, empty/0, insert/2, inorder/1},
 { ...                                      % bridge rules of LNat
  empty -> BST.empty .                      % bridge rules of BST
  insert(N,T1) -> BST.insert(N,T1) .
  inorder(T1) -> BST.inorder(T1) .

  ...                                       % hidden part of LNat
  BST.empty -> BST.nil .                    % hidden part of BST
  BST.insert(N,BST.nil) -> BST.mktree(N,BST.nil,BST.nil)
                                 <= BST.isnat(N) >< true .
  BST.insert(N,BST.mktree(M,T1,T2)) -> BST.mktree(M,T1,T2)
                                 <= N >< M, BST.isnat(N) >< true .
  BST.insert(N,BST.mktree(M,T1,T2)) -> BST.mktree(M,BST.insert(N,T1),T2)
                                 <= N NBST.< M >< true .
  BST.insert(N,BST.mktree(M,T1,T2)) -> BST.mktree(M,T1,BST.insert(N,T2))
                                 <= M MBST.< N >< true .
  BST.inorder(BST.nil) -> [] .
  BST.inorder(BST.mktree(M,T1,T2)) -> BST.inorder(T1) BST.++ [M|BST.inorder(T2)]
                                 <= BST.isnat(M) >< true .}>
```

We can use this module, only using the exported signature and visible constructor symbols, as is done in the following module for sorting lists:

```
LSort =
<{empty/0, insert/2, inorder/1},
 {listTotree/1, lsort/1},
 {listTotree([]) -> empty .
  listTotree([N|L]) -> insert(N,listTotree(L)) .
  lsort(L) -> inorder(listTotree(L)) .} >
```

The behavior of a structured module $\mathscr{P} =< \sigma_p, \sigma_e, \mathscr{R}_V \cup \mathscr{R}_B \cup \mathscr{R}_H >$ with hidden constructor symbols w.r.t. the visible signature can be expressed with the aid of the algebra transformer $\mathscr{U}_{\mathscr{P}}\colon \mathbf{CTAlg}_\Sigma \to \mathbf{CTAlg}_\Sigma$ defined, for each $\mathscr{A}$, as $\mathscr{U}_{\mathscr{P}}(\mathscr{A}) = \mathscr{T}_{\mathscr{R}_V \cup \mathscr{R}_B}(\mathscr{T}^{\omega}_{\mathscr{R}_H}(\bot_{\overline{\Sigma}}) \sqcup \overline{\mathscr{A}})|_\Sigma$, where now $\overline{\mathscr{A}}$ is the extension of $\mathscr{A}$ to an algebra of $\mathbf{CTAlg}_{\Omega\times\Sigma}$ obtained by adding functions $\mathscr{P}.f^{\overline{\mathscr{A}}}$, defined as $\mathscr{P}.f^{\overline{\mathscr{A}}}(\bar{t}) = \langle\bot\rangle$, for each $f/n \in FS_\Sigma$ and $\mathscr{P} \in \Omega$, and defining $f^{\overline{\mathscr{A}}}(\bar{t}) = f^{\mathscr{A}}(\bar{t}^*)$ where tuple $\bar{t}^*$ is obtained from $\bar{t}$ by changing each term beginning with a labeled constructor term for $\bot$, for each $f/n \in FS_\Sigma$, and $\mathscr{B}|_\Sigma$ means the reduct of the algebra $\mathscr{B} \in \mathbf{CTAlg}_{\Omega\times\Sigma}$ obtained by restricting the carrier to $\mathbf{CTerm}_\bot$ and forgetting all functions denoting labeled function symbols.

Obviously, the representation of the closure w.r.t. the functional signature is a particular case of closure hiding a set of constructor symbols when this set is empty.

9 Discussion

Research in component-based software development is currently becoming a very active area for the logic programming community. In fact, we can find several proposals in the field of computational logic for dealing with the design and development of large software systems. Other related fields, like functional-logic programming are now proving that the integration of logic variables and functions may increase the expressive power of a programming language. A number of attempts are being made in this direction (Hanus, 1994, 1998) to achieve a consensus on the characteristics a functional-logic language has to present.

The current work tries to contribute to all these efforts by presenting a notion of module in the context of functional-logic programming, and by providing a number of operations (satisfying some expected algebraic properties) expressive enough to model typical modularization issues like export/import relationships, hiding information, inheritance, and a sort of abstraction. We have chosen the Constructor-based Conditional Rewriting Logic (González-Moreno *et al.*, 1999) to develop our proposal and, in this context, we have explored a rather wide range of semantics for program modules and we have studied some of their relevant properties, in particular, those concerning compositionality and full abstraction w.r.t. the observation function $Ob(\mathscr{P}) = \mathscr{M}_{\mathscr{P}}$ and the set $\{\cup, \overline{(\cdot)}^{\sigma}, (\cdot)\setminus\sigma, \rho(\cdot)\}$ of module operations.

Although these features are interesting enough from a theoretical point of view, they present a special significance when module reusing, module refining or module transforming are involved. The least model semantics, $\{\![\cdot]\!\}_{LM}$, is a fully abstract semantics, which is only compositional w.r.t. $\{\overline{(\cdot)}^{\sigma}, \rho(\cdot)\}$, but only for injective function renamings ρ. On the contrary, the $\mathscr{T}$-semantics, $\{\![\cdot]\!\}_{T}$, is compositional (w.r.t. all operations), but is not fully abstract. The third proposal, the loose model-theoretic semantics, $\{\![\cdot]\!\}_{M}$, is also compositional (except for the deletion operation), although the full abstraction property is not satisfied. A fully abstract semantics, $\{\![\cdot]\!\}_{CM}$, may be obtained by considering a consistency property on term algebras, which is also compositional w.r.t. the union, closure and renaming operations. To recover the compositionality w.r.t. deletion we need a finer semantics able to capture the 'independent' meaning of each function in a module; this is the case of the deletion semantics, $\{\![\cdot]\!\}_{D}$, which still is fully abstract and compositional w.r.t. all operations. We have also studied the $(\mathscr{T} \sqcup Id)$-semantics, $\{\![\cdot]\!\}_{T\sqcup I}$, but we have not included this study in this paper because it exhibits the same properties as the $\mathscr{T}$-semantics. Table 1 summarizes the properties satisfied by each one of the analyzed semantics.

It is possible to establish a semantics hierarchy ranging from the model-theoretic semantics to the $\mathscr{T}$-semantics on the basis of the order $\equiv_T \sqsubseteq \equiv_{T\sqcup Id} \sqsubseteq \equiv_D \sqsubseteq \equiv_{CM} \sqsubseteq \equiv_C$ for the equivalence relationships induced by these semantics, where they are ordered upon their strength. The $\mathscr{T}$-equivalence relation, $\equiv_T$, is the strongest one, and it is contained obviously into the $(\mathscr{T} \sqcup Id)$-equivalence relation, $\equiv_{T\sqcup Id}$. Taking into account that this equivalence relation is compositional but not fully abstract, it will be contained in $\equiv_D$, which is also contained in the consistent term-model equivalence, $\equiv_{CM}$. Obviously, the least term-model equivalence, $\equiv_{LM}$, is the weakest one.

To establish some conclusions about the compositionality and the full abstraction of all these semantics, we are going to discuss the information exhibited in Table 1. In this table, we can observe a sort of dependency between fulfilling compositionality/full abstraction and the strength of the equivalence relationship defined by the semantics, in such a way that the strongest ones are compositional whereas the weakest ones are fully abstract. The best semantics must be an intermediate semantics satisfying both properties; in our case, the semantics $\{\![\cdot]\!\}_{D}$. A similar

Table 1. *Compositionality (C) and full abstraction (FA)*

	$\cup, \overline{(\cdot)}^{\sigma}, (\cdot)\backslash\sigma, \rho(\cdot)$	$\cup, \overline{(\cdot)}^{\sigma}, \rho(\cdot)$	$\overline{(\cdot)}^{\sigma}, \rho(\cdot)$	$\overline{(\cdot)}^{\sigma}$
$\{[\cdot]\}_{T}$	C	C	C	C
$\{[\cdot]\}_{T\sqcup I}$	C	C	C	C
$\{[\cdot]\}_{D}$	FA C	C	C	C
$\{[\cdot]\}_{CM}$	FA	FA C	C	C
$\{[\cdot]\}_{LM}$	FA	FA	FA	FA C

study was already made by Brogi and Turini (1995) in the field of logic programming, but he did not deal with variables, and avoided the complexity inherent to the non-ground term algebras. Another difference (apart from the context) with respect to the current work is the set of operations we are considering, which does not coincide with the set of inter-module operations defined by Brogi. One of the most significative operations described by him is the intersection of programs. This operation makes the $(\mathcal{T} \sqcup Id)$-semantics compositional and fully abstract in a logic programming context. However, the difficult justification of this operation in our framework (the functional-logic programming paradigm) has inclined us to think in an alternative: the deletion operation. We believe that this operation is more natural (as a composing mechanism) than program intersection. This has an inconvenience: the $(\mathcal{T} \sqcup Id)$-semantics is not fully abstract (although it is compositional) w.r.t. our operations. In fact, the intersection of programs is a very powerful tool to distinguish programs (more than the deletion operation), and it can be used to delete a single rule, whereas our deletion operation only can be used to delete a whole set of rules defining a function. Nevertheless, we have found a fully abstract and compositional semantics, also for the deletion operation, which completes the results provided by this work.

Acknowledgements

We would like to thank Mario Rodríguez-Artalejo and Ana Gil Luezas for their helpful comments and suggestions on the initial versions of this work. We would also like to thank Narciso Martí-Oliet for their insightful comments and suggestions, that greatly helped us improving the quality and presentation of the paper. This work has been partially supported by the Spanish project TIC98-0445-C03-03.

References

Abramsky, S. and Jung, A. (1994) Domain theory. *Handbook of Logic in Computer Science*, **3**, 1–168. Oxford University Press.

Apt, K. (1990) Logic programming. *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, pp. 493–574. Elsevier.

Apt, K. (1996) *From Logic Programming to Prolog*. Prentice-Hall.

Bergstra, J. A., Heering, J. and Klint, P. (1990) Module algebra. *J. ACM*, **37**(2), 335–372.

Brogi, A. (1993) *Program Construction in Computational Logic*. PhD Thesis TD-2/93, University of Pisa-Genova-Udine.

Brogi, A., Mancarella, P., Pedreschi, D. and Turini, F. (1994) Modular Logic programming. *ACM Trans. Program. Lang. & Syst.* **16**(4), 1361–1398.

Brogi, A. and Turini, F. (1995) Fully abstract compositional semantics for an algebra of logic programs. *Theor. Comput. Sci.* **149**(2), 201–229.

Bugliesi, M., Lamma, E. and Mello, P. (1994) Modularity in logic programming. *J. Logic Program.* **19**(20), 443–502.

Durán, F. (1999) *A Reflective Module Algebra with Applications to the Maude Language*. PhD Thesis, University of Málaga.

Ehrig, H. and Mahr, B. (1990) *Fundamentals of Algebraic Specification 2. Module Specifications and Constraints*. Springer-Verlag.

Falaschi, M., Levi, G., Martelli, M. and Palamidessi, C. (1993) A model-theoretic reconstruction of the operational semantics of logic programs. *Infor. & Computation*, **102**(1), 86–113.

Goguen, J. A. and Burstall, C. R. M. (1992) Institutions: abstract model theory for specification and programming. *J. ACM*, **39**(1), 95–146.

González-Moreno, J. C. (1994) *Programación Lógica de Orden Superior con Combinadores*. PhD Thesis, Universidad Complutense de Madrid.

González-Moreno, J. C., Hortalá-González, M. T., López-Fraguas, F. J. and Rodríguez-Artalejo, M. (1999) An approach for declarative programming based on a rewriting logic. *J. Logic Programl.* **40**(1), 47–88.

Hanus, M. (1994) The integration of functions into logic programming. A survey. *J. Logic Program.* **19**(20), 583–628.

Hanus, M. (ed.) (2000) *Curry. An Integrated Functional Logic Language*. Draft (available at: www.informatik.uni_kiel.de/~mh/curry/report.html).

Hussmann, H. (1993) *Non-determinism in Algebraic Specifications and Algebraic Programs*. Birkäuser Verlag.

Mancarella, P. and Pedreschi, D. (1988) An algebra of logic programs. In: Kowalski, R. A. and Bowen, A. (eds.) *Proc. 5th International Conference on Logic Programming*, pp. 1006–1023. MIT Press.

Meseguer, J. (1992) Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.* **96**, 73–155.

Miller, D. (1986) A theory of modules in logic programming. *Proc. of Symposium of Logic Programming*, pp. 106–114.

Molina-Bravo, J. M. and Pimentel, E. (1997) Modularity in functional-logic programming. *Proc. 14th International Conference on Logic Programming*, pp. 183–197. MIT Press.

Molina-Bravo, J. M. (2000) *Modularidad en Programación Lógico-Funcional de Primer Orden*. PhD Thesis, University of Málaga.

Möller, B. (1985) On the algebraic specification of infinite objects – ordered and continuous models of algebraic types. *Acta Informatica*, **22**, 537–578.

Orejas, F., Pino, E. and Ehrig, H. (1997) Institutions for logic programming. *Theor. Comput. Sci.* **173**, pp. 485–511.

Orejas, F. (1999) Structuring and modularity. In: Astesiano, E., Kreowski, H.-J. and Krieg-Brückner, B. (eds.), *Algebraic Foundations of Systems Specification*, pp. 159–200. Springer-Verlag.

Wirsing, M. (1990) Algebraic specification. *Handbook of Theoretical Computer Science, Vol. B*, pp. 675–788. Elsevier.